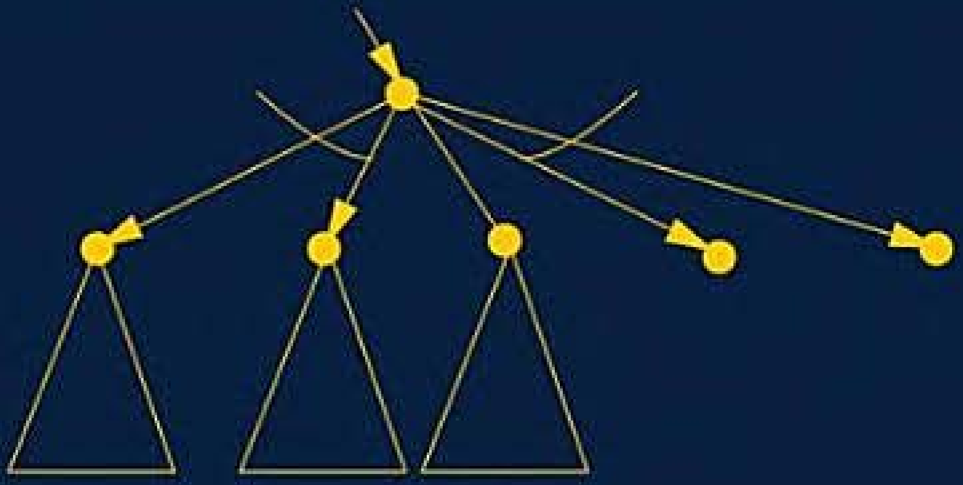
Marco Bernardo
Valérie Issarny (Eds.)

Tutorial

LNCS 6659

Formal Methods for Eternal Networked Software Systems

11th International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM 2011
Bertinoro, Italy, June 2011, Advanced Lectures



Springer

Marco Bernardo   Valérie Issarny (Eds.)

Formal Methods for Eternal Networked Software Systems

11th International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM 2011
Bertinoro, Italy, June 13-18, 2011
Advanced Lectures

Springer

Volume Editors

Marco Bernardo
Università di Urbino "Carlo Bo"
Dipartimento di Scienze di Base e Fondamenti
Piazza della Repubblica 13, 61029 Urbino, Italy
E-mail: bernardo@sti.uniurb.it

Valérie Issarny
INRIA Paris - Rocquencourt
Domaine de Voluceau, B.P. 105
78153 Le Chesnay, France
E-mail: valerie.issarny@inria.fr

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-21454-7 e-ISBN 978-3-642-21455-4
DOI 10.1007/978-3-642-21455-4
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2011928397

CR Subject Classification (1998): D.2.4, D.3.1, F.3-4, C.3

LNCS Sublibrary: SL 2 – Programming and Software Engineering

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

This volume presents a set of papers accompanying the lectures of the 11th International School on Formal Methods for the Design of Computer, Communication and Software Systems (SFM).

This series of schools addresses the use of formal methods in computer science as a prominent approach to the rigorous design of the above-mentioned systems. The main aim of the SFM series is to offer a good spectrum of current research in foundations as well as applications of formal methods, which can be of help for graduate students and young researchers who intend to approach the field.

SFM 2011 was devoted to formal methods for eternal networked software systems and covered several topics including formal foundations for the interoperability of software systems, application-layer and middleware-layer dynamic connector synthesis, interaction behavior monitoring and learning, and quality assurance of connected systems. The school was held in collaboration with the researchers of the EU-funded projects CONNECT and ETERNALS.

This volume comprises 15 articles organized into six parts: (i) architecture and interoperability, (ii) formal foundations for connectors, (iii) connector synthesis, (iv) learning and monitoring, (v) dependability assurance, and (vi) trustworthy eternal systems via evolving software.

The paper by Blair, Paolucci, Grace, and Georgantas examines the issue of interoperability in complex distributed systems by focussing on middleware solutions that are intrinsically based on semantic meaning and advocates a dynamic approach to interoperability based on the concept of emergent middleware. Grace, Georgantas, Bennaceur, Blair, Chauvel, Issarny, Paolucci, Saadi, Souville, and Sykes illustrate how the CONNECT architecture tackles the interoperability problem for heterogeneous systems by observing the networked systems in action, learning their behavior, and then dynamically generating mediator software that will connect the systems.

The paper by Forejt, Kwiatkowska, Norman, and Parker provides an introduction to probabilistic model checking of Markov decision processes and its applications to performance and dependability analysis of networked systems, communication protocols, and randomized distributed algorithms. Baier, Klein, and Klüppelholz present an overview of the modeling concepts for components and connectors using the exogenous coordination languages Reo together with the underlying constraint automata framework for property verification.

The paper by Inverardi, Spalazzese, and Tivoli reports on how to automatically achieve protocol interoperability via connector synthesis by distinguishing between two notions of application-layer connectors: coordinators and mediators. Giannakopoulou and Păsăreanu review techniques for generating component interfaces automatically in order to cope with the fact that the satisfaction of certain properties may depend on the context in which a component will be dynamically introduced. The paper by Issarny, Bennaceur, and Bromberg deals with middleware interoperability by discussing an approach to the dynamic synthesis of emergent connectors that mediate the interaction protocols executed by networked systems from application down to middleware layers.

Steffen, Howar, and Merten give an introduction to active learning of Mealy machines, which is characterized by the alternation of an exploration phase – during which membership queries are used to construct hypothesis models of a system under test – and a testing phase – during which equivalence queries are used to compare hypothesis models with the actual system – until a valid model of the target system is produced. The paper by Tretmans's presents model-based testing, in which test cases are algorithmically generated from a model specifying the required behavior of a system, and test-based modeling or automata learning, which aims at automatically generating a model from test observations, and shows that test coverage in model-based testing and precision of learned models turn out to be two sides of the same coin. Jonsson's paper is about generating models of communication system components from observations of their external behavior and illustrates how to adapt existing techniques to include data parameters in messages and states.

The paper by Bertolino, Calabró, Di Giandomenico, and Nostro deals with the dependability and performance evaluation of dynamic and evolving systems by means of a framework that can be used off-line for system design and online for continuously monitoring system behavior and detecting possible issues arising at run time. Costa, Issarny, Martinelli, Matteucci, and Saadi investigate security and trust as two complementary perspectives on the problem of the correct interaction among software components and propose an approach called security by contract with trust, in which the level of trust measures the adherence of the application to its contract.

The paper by Clarke, Diakov, Hähnle, Johnsen, Schaefer, Schäfer, Schlatte, and Wong describes HATS, an abstract behavioral modeling language for highly configurable distributed systems that supports spatial and temporal variability. Moschitti's paper introduces kernel methods designed within the statistical learning theory in order to overcome the concrete limitations of logic/rule-based approaches to the semantic modeling of the behavior of complex systems. Jürjens, Ochoa, Schmidt, Marchal, Houmb, and Islam recall the UMLsec approach to model-based security, which supports the system specification and design phases as well as maintaining the needed levels of security even through later software evolution.

We believe that this book offers a useful view of what has been done and what is going on worldwide in the field of eternal networked software systems. We wish to thank all the speakers and all the participants for a lively and fruitful school. We also wish to thank the entire staff of the University Residential Center of Bertinoro for the organizational and administrative support.

June 2011

Marco Bernardo
Valérie Issarny

Table of Contents

Part V: Dependability Assurance

Interoperability in Complex Distributed Systems

Gordon S. Blair[1], Massimo Paolucci[2], Paul Grace[1], and Nikolaos Georgantas[3]

[1] School of Computing and Communications, Lancaster University, UK
{gordon,gracep}@comp.lancs.ac.uk
[2] Laboratories Europe GmbH, Munich, Germany
paolucci@docomolab-euro.com
[3] INRIA, CRI Paris-Rocquencourt, France
nikolaos.georgantas@inria.fr

Abstract. Distributed systems are becoming more complex in terms of both the level of heterogeneity encountered coupled with a high level of dynamism of such systems. Taken together, this makes it very difficult to achieve the crucial property of interoperability that is enabling two arbitrary systems to work together relying only on their declared service specification. This chapter examines this issue of interoperability in considerable detail, looking initially at the problem space, and in particular the key barriers to interoperability, and then moving on to the solution space, focusing on research in the middleware and semantic interoperability communities. We argue that existing approaches are simply unable to meet the demands of the complex distributed systems of today and that the lack of integration between the work on middleware and semantic interoperability is a clear impediment to progress in this area. We outline a roadmap towards meeting the challenges of interoperability including the need for integration across these two communities, resulting in middleware solutions that are intrinsically based on semantic meaning. We also advocate a dynamic approach to interoperability based on the concept of emergent middleware.

Keywords: Interoperability, complex distributed systems, heterogeneity, adaptive distributed systems, middleware, semantic interoperability.

1 Introduction

Complex pervasive systems are replacing the traditional view of homogenous distributed systems, where domain-specific applications are individually designed and developed upon domain-specific platforms and middleware, for example, Grid applications, Mobile Ad-hoc Network applications, enterprise systems and sensor networks. Instead, these technology-dependent islands are themselves dynamically composed and connected together to create richer interconnected structures, often referred to as systems of systems. While there are many challenges to engineering such complex distributed systems, a central one is 'interoperability', i.e., the ability for one or more systems to: connect, understand and exchange data with one another for a given purpose. When considering interoperability there are two key properties to deal with:

- *Extreme heterogeneity*. Pervasive sensors, embedded devices, PCs, mobile phones, and supercomputers are connected using a range of networking solutions, network protocols, middleware protocols, and application protocols and data types. Each of these can be seen to add to the plethora of technology islands, i.e., systems that cannot interoperate.
- *Dynamic and spontaneous communication*. Connections between systems are not made until runtime; no design or deployment decision, e.g., the choice of middleware, can inform the interoperability solution.

We highlight in this chapter the important dimensions that act as a barrier to interoperability; these consist of differences in: the data formats and content, the application protocols, the middleware protocols and the non-functional properties. We then investigate state-of-the-art solutions to interoperability from the middleware and the semantic web community. This highlights that the approaches so far are not fit for purpose, and importantly that the two communities are disjoint from one another. Hence, we advocate that the two fields embrace each other's results, and that from this, fundamentally different solutions will emerge in order to drop the interoperability barrier.

2 Interoperability Barriers: Dimensions of Heterogeneity

2.1 Data Heterogeneity

Different systems choose to represent data in different ways, and such data representation heterogeneity is typically manifested at two levels. The simplest form of data interoperability is at the syntactic level where two different systems may use two very different formats to express the same information. Consider a vendor application for the sale of goods; one vendor may price an item using XML, while another may serialize its data using a Java-like syntax. So the simple information that the item costs £1 may result in the two different representations as shown in Fig. 1(a).

```
<price>
    <value> 1 </value>
    <currency> GBP </currency>
</price>
```

```
price(1,GBP)
```

a) Representing price in XML and tuple data

```
<price>
    <value> 1 </value>
    <currency> GBP </currency>
</price>
```

```
<cost>
  <amount> 1 </ amount >
  <denomination> £</ denomination >
</cost>
```

b) Heterogeneous Currency Data

Fig. 1. Examples of Data Heterogeneity

Aside from the syntactic level interoperability, there is a greater problem with the "*meaning*" of the tokens in the messages. Even if the two components use the same syntax, say XML, there is no guarantee that the two systems recognize all the nodes in the parsing trees or even that the two systems interpret all these nodes in a consistent way. Consider the two XML structures in the example in Fig. 1(b). Both structures are in XML and they (intuitively) carry the same meaning. Any system that recognizes the first structure will also be able to parse the second one, but will fail to recognize the similarity between them unless the system realizes that *price≡cost*, that *value≡amount*, that *currency≡denomination* and of course that *GBP≡£* (where ≡ means equivalent). The net result of using XML is that both systems will be in the awkward situation of parsing each other's message, but not knowing what to do with the information that they just received.

The deeper problem of data heterogeneity is the semantic interoperability problem whereby all systems provide the same interpretation to data. The examples provided above, show one aspect of data interoperability, namely the recognition that two different labels represent the same object. This is in the general case an extremely difficult problem which is under active research [1], though in many cases it can receive a simple pragmatic solution by forcing the existence of a shared dictionary. But the semantic interoperability problem goes beyond the recognition that two labels refer to the same entity. Ultimately, the data interoperation problem is to guarantee that all components of the system share the same understanding of the data transmitted, where the same understanding means that they have consistent semantic representations of the data.

2.2 Middleware Heterogeneity

Developers can choose to implement their distributed applications and services upon a wide range of middleware solutions that are now currently available. In particular, these consist of heterogeneous *discovery* protocols which are used to locate or advertise the services in use, and *heterogeneous interaction* protocols which perform the direct communication between the services. Fig. 2 illustrates a collection implemented upon these different technologies. Application 1 is a mobile sport news application, whereby news stories of interest are presented to the user based on their current location. Application 2 is a jukebox application that allows users to select and play music on an audio output device at that location. Finally, application 3 is a chat application that allows two mobile users to interact with one another. In two locations (a coffee bar and a public house) the same application services are available to the user, but their middleware implementations differ. For example, the Sport News service is implemented as a publish-subscribe channel at the coffee bar and as a SOAP service in the pubic house. Similarly, the chat applications and jukebox services are implemented using different middleware types. The service discovery protocols are also heterogeneous, i.e., the services available at the public house are discoverable using SLP and the services at the coffee bar can be found using both UPnP and SLP. For example, at the coffee bar the jukebox application must first find its corresponding service using UPnP and then use SOAP to control functionality. When it moves to the public house, SLP and CORBA must be used.

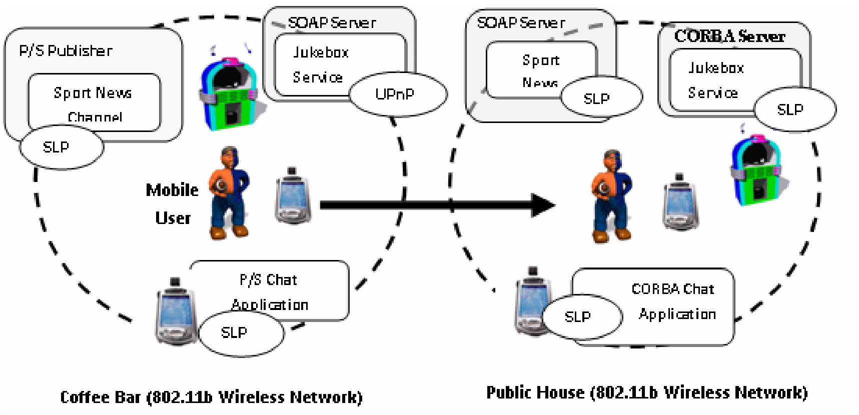


Fig. 2. Legacy services implemented using heterogeneous middleware

2.3 Application Heterogeneity

Interoperability challenges at the application level might arise due to the different ways the application developers might choose to implement the program functionality, including different use of the underlying middleware. As a specific example, a merchant application could be implemented using one of two approaches for the consumer to obtain information about his wares:

— A single `GetInfo()` remote method, which returns the information about the product price and quantity available needed by the consumer.
— Two separate remote methods `GetPrice()`, and `GetQuantity()` returning the same information.

A client developer can then code the consumer using either one of the approaches described above, and this would lead to different sequences of messages between the consumer and merchant. Additionally, application level heterogeneity can also be caused due to the differences between the underlying middlewares. For example, when using a Tuple Space, the programmer can use the rich search semantics provided by it, which are not available in other types of middleware, e.g., for RPC middleware a Naming Service or discovery protocol must then be used for equivalent capabilities.

2.4 Non-functional Heterogeneity

Distributed systems have non-functional properties that must also be considered if interoperability is to be achieved. That is, two systems may be able to overcome all of the three prior barriers and functionally interoperate, but if the solution does not satisfy the non-functional requirements of each of the endpoints then it cannot be considered to have achieved full interoperability. For example, peers may have different requirements for the latency of message delivery; if the client requires that messages be delivered within 5ms and the server can only achieve delivery in 10ms then interoperability is not satisfying the solution. Similarly, two systems may employ different security protocols; the interoperability solution must ensure that the security requirements of both systems are maintained.

3 Middleware Solutions to Interoperability

3.1 Introduction

Tanenbaum and Van Steen define interoperability as:

> *"the extent by which* two *implementations of systems or components from different manufacturers can co-exist and work together by merely relying on each other's services as specified by a common standard."* [2].

Achieving such interoperability between independently developed systems has been one of the fundamental goals of middleware researchers and developers. This section traces these efforts looking at traditional middleware that seek a common standard/ platform for the entire distributed system (section 3.2), interoperability platforms that recognize that middleware heterogeneity is inevitable and hence allows clients to communicate with a given middleware as dynamically encountered (section 3.3), software bridges that support the two-way translation between different middleware platforms (section 3.4), transparent interoperability solutions that go beyond interoperability platforms by allowing two legacy applications to transparently communicate without any change to these applications (section 3.5), and finally the logical mobility approach that overcomes heterogeneity by migrating applications and services to the local environment, assuming that environment has the mechanisms to interpret this code, e.g. through an appropriate virtual machine.

3.2 Traditional Middleware

The traditional approach to resolving interoperability using middleware standards and platforms is to advocate that all systems are implemented upon the same middleware technology; this pattern is illustrated in Fig. 3 and is equivalent to native spoken language interoperability where the speakers agree in advance upon one language to speak. There are many different middleware styles that follow this pattern of interoperability, and it is important to highlight that these actually contribute to the interoperability problem, i.e., the different styles and specific implementations do not interoperate; to illustrate this point the following is a list of the most commonly used solution types:

— *RPC/Distributed Objects.* Distributed Objects (e.g. CORBA [3] and DCOM [4]) are a communication abstraction where a distributed application is decomposed into objects that are remotely deployed and communicate and co-ordinate with one another. The abstraction is closely related to the well-established methodology of object orientation, but rather than method invocations between local objects, distributed objects communicate using remote method invocations; where a method call and parameters are marshalled and sent across the network and the result of the method is returned synchronously to the caller. This is similar to the style of communication employed in remote procedure calls (RPC) e.g. SunRPC [5].

- *Message-based.* Messaging applications differ from RPC in that they provide a one-way, asynchronous exchange of data. This can either be i) direct between two endpoints e.g. SOAP messaging, or ii) involve an intermediary element such as a message queue that allows the sender and receiver to be decoupled in time and space i.e. both do not need to be connected directly or at the same time. Examples of message queue middleware are MSMQ [6] and the Java Messaging Service (JMS)[1].
- *Publish-Subscribe* is an alternative messaging abstraction where the producers and consumers of messages are anonymous from one another. Consumers subscribe for content by publishing a subscription (this can be topic-based, i.e., based upon the type of the message, or content-based i.e. the filter is fine-grained towards the content of each message); and publishers then send out messages/events. Brokers are intermediary systems that are deployed in the network or at the edge which match messages to subscriptions. A match then requires the event to be delivered to the application. Notable examples of Publish-Subscribe middleware are SIENA [7] and JMS.
- *Tuple Spaces*. The Linda platform [8] originated the concept of tuple spaces as a shared-memory approach for the coordination of systems. Clients can write and read data tuples into a shared space, where a tuple is a data element much like a database record. Tuple space middleware often differ in how the tuple space is deployed e.g. enterprise solutions, such as TSpaces [9], use a central enterprise server to host the tuple space for clients to connect to, while L^{2}imbo [10] and LiME[11] distribute the tuple space evenly among peers.

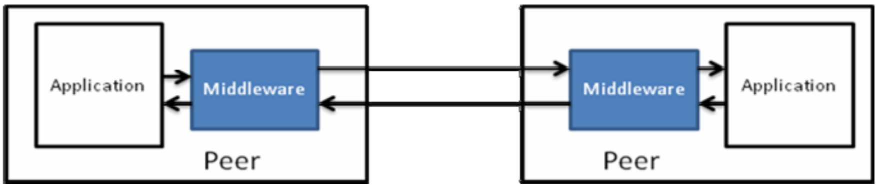


Fig. 3. Interoperability pattern utilised by traditional middleware

These technologies resolve interoperability challenges to different extents, the majority focusing on interoperation between systems and machines with heterogeneous hardware and operating systems, and applications written in different programming languages. Hence, this pattern works well for distributed systems where the parties and technologies are known in advance and can be implemented using a common middleware choice. However, for pervasive and dynamic environments where systems interact spontaneously this approach is infeasible (every application would be required to be implemented upon the same middleware). In the more general sense of achieving universal interoperability and dynamic interoperability between spontaneous communicating systems they have failed. Within the field of

[1] http://www.oracle.com/technetwork/java/jms/index.html

distributed software systems, any approach that assumes a common middleware or standard is destined to fail due to the following reasons:

1. A one size fits all standard/middleware cannot cope with the extreme heterogeneity of distributed systems, e.g. from small scale sensor applications through to large scale Internet applications. CORBA and Web Services [12] both present a common communication abstraction i.e. distributed objects or service orientation. However, the list of diverse middleware types already illustrates the need for heterogeneous abstractions.
2. New distributed systems and application emerge fast, while standards development is a slow, incremental process. Hence, it is likely that new technologies will appear that will make a pre-existing interoperability standard obsolete, c.f. CORBA versus Web Services (neither can talk to the other).
3. Legacy platforms remain useful. Indeed, CORBA applications remain widely in use today. However, new standards do not typically embrace this legacy issue; this in turn leads to immediate interoperability problems.

3.3 Interoperability Platforms

Fig. 4 illustrates the pattern employed by *interoperability platforms*, which can be seen to follow the spoken language translation approach of the person speaking another person's language. Interoperability platforms provide a middleware-agnostic technology for client, server, or peer applications to be implemented directly upon in order to guarantee that the application can interoperate with all services irrespective of the middleware technologies they employ. First, the interoperability platform presents an API for developing applications with. Secondly, it provides a substitution mechanism where the implementation of the protocol to be translated to, is deployed locally by the middleware to allow communication directly with the legacy peers (which are simply legacy applications and their middleware). Thirdly, the API calls are translated to the substituted middleware protocol. A key feature of this approach is that it does not require reliance on interoperability software located elsewhere, e.g., a remote bridge, an infra-structure server, or the corresponding endpoint; this makes it ideal for infra-structureless environments. For the particular use case, where you want a client application to interoperate with everyone else, interoperability platforms are a powerful approach. These solutions rely upon a design time choice to develop applications upon the interoperability platforms; therefore, they are unsuited to other interoperability cases, e.g., when two applications developed upon different legacy middleware want to interoperate spontaneously at runtime. We now discuss three key examples of interoperability platforms.

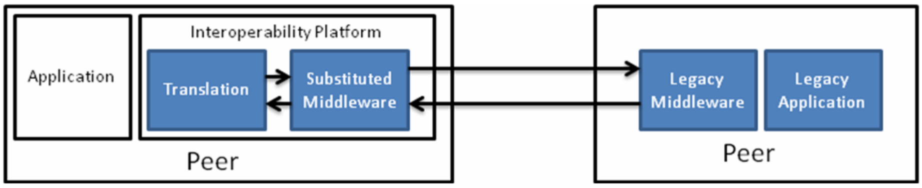


Fig. 4. Interoperability pattern utilised by interoperability platforms

Universally Interoperable Core (UIC) [13] was an early solution to the middleware interoperability problem; in particular it was an adaptive middleware whose goal was to support interactions from a mobile client system to one or more types of distributed object solutions at the server side, e.g., CORBA, SOAP and Java RMI. The UIC implementation was based upon the architectural strategy pattern of the dynamicTAO system [14]; namely, a skeleton of abstract components that form the base architecture is then specialised to the specific properties of particular middleware platforms by adding middleware specific components to it (e.g. a CORBA message marshaller and demarshaller).

ReMMoC [15] is an adaptive middleware developed to ensure interoperability between mobile device applications and the available services in their local environment. Here, two phases of interoperability are important: i) discovery of available services in the environment, and ii) interaction with a chosen service. The solution is a middleware architecture that is employed on the client device for applications to be developed upon. It consists of two core frameworks. A service discovery framework is configured to use different service discovery protocols in order to discover services advertised by those protocols; a complete implementation of each protocol is plugged into the framework. Similarly, a binding framework allows the interaction between services by plugging-in different binding type implementations, e.g., an IIOP client, a publisher, a SOAP client, etc.

The **Web Services Invocation Framework** (WSIF) [16] is a Java API, originating at IBM and now an Apache release, for invoking Web Services irrespective of how and where these services are provided. Its fundamental goal is to achieve a solution to better client and Web Service interoperability by freeing the Web Services Architecture from the restrictions of the SOAP messaging format. WSIF utilises the benefits of discovery and description of services in WSDL, but applied to a wider domain of middleware, not just SOAP and XML messages. The structure of WSDL allows the same abstract interface to be implemented by multiple message binding formats, e.g., IIOP and SOAP; to support this, the WSDL schema is extended to understand each format. The core of the framework is a pluggable architecture into which providers can be placed. A provider is a piece of code that supports each specific binding extension to the WSDL description, i.e., the provider uses the specification to map an invoked abstract operation to the correct message format for the underlying middleware.

3.4 Software Bridges

Software bridges enable communication between different middleware environments. Hence, clients in one middleware domain can interoperate with servers in another middleware domain. The bridge acts as a one-to-one mapping between domains; it will take messages from a client in one format and then marshal this to the format of the server middleware; the response is then mapped to the original message format. Fig. 5 illustrates this pattern, which can be seen as equivalent to employing a translator to communicate between native speakers. Many bridging solutions have been produced between established commercial platforms The OMG has created the DCOM/CORBA Inter-working specification [17] that defines the bi-directional mapping between DCOM and CORBA and the locations of the bridge in the process.

SOAP2CORBA[2] is an open source implementation of a fully functional bi-directional SOAP to CORBA Bridge. While a recognised solution to interoperability, bridging is infeasible in the long term as the number of middleware systems grow, i.e., due to the effort required to build direct bridges between all of the different middleware protocols.

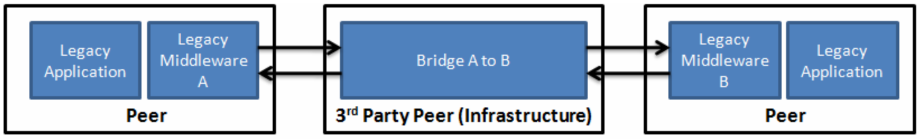


Fig. 5. Interoperability pattern utilised by Software Bridges

Enterprise Service Buses (ESB) can be seen as a special type of software bridge; they specify a service-oriented middleware with a message-oriented abstraction layer atop different messaging protocols (e.g., SOAP, JMS, SMTP). Rather than provide a direct one-to-one mapping between two messaging protocols, a service bus offers an intermediary message bus. Each service (e.g. a legacy database, JMS queue, Web Service etc.) maps its own message onto the bus using a piece of code, to connect and map, deployed on the peer device. The bus then transmits the intermediary messages to the corresponding endpoints that reverse the translation from the intermediary to the local message type. Hence traditional bridges offer a 1-1 mapping; ESBs offer an N-1-M mapping. Example ESBs are Artix[3] and IBM Websphere Message Broker[4].

Bridging solutions have shown techniques whereby two protocols can be mapped onto one another. These can either use a one-to-one mapping or an intermediary bridge; the latter allowing a range of protocols to easily bridge between one another. This is one of the fundamental techniques to achieve interoperability. Furthermore, the bridge is usually a known element that each of the end systems must be aware of and connect to in advance-again this limits the potential for two legacy-based applications to interoperate.

3.5 Transparent Interoperability

In transparent interoperability neither legacy implementation is aware of the encountered heterogeneity, and hence legacy applications can be made to communicate with one another. Fig. 6 shows the key elements of the approach. Here, the protocol specific messages, behaviour and data are captured by the interoperability framework and then translated to an intermediary representation (note the special case of a one-to-one mapping, or bridge is where the intermediary is the corresponding protocol); a subsequent mapper then translates from the intermediary to the specific legacy middleware to interoperate with. The use of an intermediary means that one middleware can be mapped to any other by developing these two elements only (i.e. a direct mapping to every other protocol is not required). Another

[2] http://soap2corba.sourceforge.net
[3] http://web.progress.com/en/sonic/artix-index.html
[4] http://www-01.ibm.com/software/integration/wbimessagebroker/

difference to bridging is that the peers are unaware of the translators (and no software is required to connect to them, as opposed to connecting applications to 'bridges'). There are a number of variations of this approach, in particular where the two parts of the translation process are deployed. They could be deployed separately or together on one or more of the peers (but in separate processes transparent to the application); however, they are commonly deployed across one or more infrastructure servers.

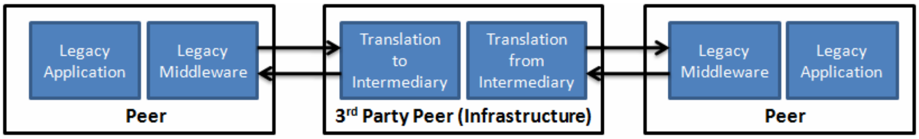

Legacy Application
Legacy Middleware
Peer
Translation to Intermediary
Translation from Intermediary
3rd Party Peer (Infrastructure)
Legacy Middleware
Legacy Application
Peer

Fig. 6. Interoperability pattern utilised by Transparent Interoperability Solutions

There are four important examples of transparent interoperability solutions:

— *The INteroperable DIscovery System for networked Services (**INDISS**)* system [18] is a service discovery middleware based on event-based parsing techniques to provide service discovery interoperability in home networked environments. INDISS subscribes to several SDP multicast groups and listens to their respective ports. To then process the incoming raw data flow INDISS uses protocol specific parsers, which are responsible for translating the data into a specific message syntax (e.g. SLP) and then extracting semantic concepts (e.g. a lookup request) into an intermediary event format. Events are then delivered to composers that translate this event to the protocol specific message of (e.g. UPnP) the protocol to interoperate with.

— ***uMiddle*** [19] is a distributed middleware infrastructure that ties devices from different discovery domains into a shared domain where they can communicate with one another through uMiddle's common protocol. To achieve interoperability uMiddle makes use of mappers and translators. Mappers function as service-level and transport-level bridges. That is, they serve as bridges that connect service discovery (e.g. SLP) and binding (e.g. SOAP) protocols to uMiddle's common semantic space. Translators project service-specific semantics into the common semantic space, act as a proxy for that service and embody any protocol and semantics that are native to the associated service.

— The *Open Service Discovery Architecture* (**OSDA**) [20] is a scalable and programmable middleware for cross-domain discovery over wide-area networks (where a domain represents a particular discovery protocol. Its motivation is the need to integrate consumers and providers across different domains irrespective of the network they belong to. OSDA assumes that discovery agents (i.e. the service registry, service consumer and service provider) are already in place. To enable cross-domain service discovery, OSDA utilizes service brokers and a peer to peer indexing overlay. Service brokers function as interfaces between the OSDA inter-domain space and the different discovery systems and are responsible for handling and processing cross-domain service registrations and requests.

— ***SeDiM*** [21] is a component framework that self-configures its behaviour to match the interoperability requirements of deployed discovery protocols, i.e., if it detects SLP and UPnP in use, it creates a connector between the two. It can be deployed as either an interoperability platform (i.e. it presents an API to develop applications that will interoperate with all discovery protocols cf. ReMMoC), or it can be utilised as a transparent interoperability solution, i.e., it can be deployed in the infrastructure, or any available device in the network and it will translate discovery functions between the protocols in the environment. SeDiM provides a skeleton abstraction for implementing discovery protocols which can then be specialised with concrete middleware. These configurations can then be 'substituted' in an interoperability platform or utilised as one side of a bridge.

Transparent interoperability solutions allow interoperability to be achieved between two legacy-based platforms; and in this sense they meet the requirements for spontaneous interoperability. However, the fundamental problem with these approaches is the Greatest Common Divisor (GCD) problem; you must identify a subset of functionality between all protocols where they match. However, as the number of protocols increases this set becomes smaller and smaller restricting what is possible.

3.6 Logical Mobility

Logical mobility is characterised by mobile code being transferred from one device and executed on another. The approach to resolve interoperability is therefore straightforward; a service advertises its behaviour and also the code to interact with it. When a client discovers the service it will download this software and then use it. Note, such an approach relies on the code being useful somewhere, i.e., it could fit into a middleware as in the substitution approach, provide a library API for the application to call, or it could provide a complete application with GUI to be used by the user. The overall pattern is shown in Fig. 7. The use of logical mobility provides an elegant solution to the problem of heterogeneity; applications do not need to know in advance the implementation details of the services they will interoperate with, rather they simply use code that is dynamically available to them at run-time. However, there are fewer examples of systems that employ logical mobility to resolve interoperability because logical mobility is the weakest of the interoperability approaches; it relies on all applications conforming to the common platform for executable software to be deployed. We now discuss two of these examples.

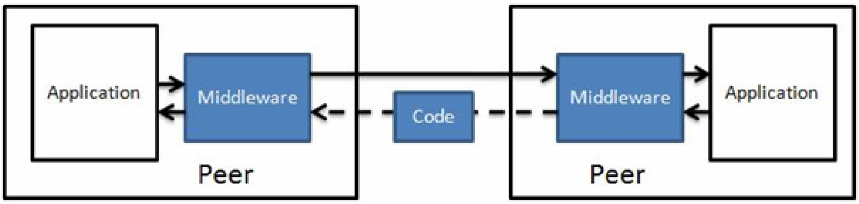


Fig. 7. Interoperability pattern utilised by Logical Mobility Solutions

SATIN [22] is a low footprint component based middleware that composes applications and the middleware itself into a set of deployable capabilities (a unit of functionality), for example, a discovery mechanism or compression algorithm. At the heart of SATIN is the ability to advertise and middleware capabilities. For example, a host uses SATIN to lookup the required application services; the interaction capabilities are then downloaded to allow the client to talk to the service.

Jini [23] is a Java based service discovery platform that provides an infrastructure for delivering services and creating spontaneous interactions between clients and services regardless of their hardware or software implementation. New services can be added to the network, old services removed and clients can discover available services all without external network administration. When an application discovers the required service, the service proxy is downloaded to their virtual machine so that it can then use this service. A proxy may take a number of forms: i) the proxy object may encapsulate the entire service (this strategy is useful for software services requiring no external resources); ii) the downloaded object is a Java RMI stub, for invoking methods on the remote service; and iii) the proxy uses a private communication protocol to interact with the service's functionality. Therefore, the Jini architecture allows applications to use services in the network without knowing anything about the wire protocol that the service uses or how the service is implemented.

4 Semantics-Based Interoperability Solutions

4.1 Introduction

The previous middleware-based solutions support interoperation by abstract protocols and language specifications. But, by and large they ignore the data heterogeneity dimension. As highlighted in Section 2.1, for two parties to interoperate it is not enough to guarantee that the data flows across, but that they both build a semantic representation of the data that is consistent across the components boundaries. The data problem has been defined in Hammer and McLeod [24] as:

> *"variations in the manner in which data is specified and structured in different components. Semantic heterogeneity is a natural consequence of the independent creation and evolution of autonomous databases which are tailored to the requirements of the application system they serve".*

Historically the problem has been well known in the database community where there is often the need to access information on different database which do not share the same data schema. More recently, with the advent of the open architectures, such as Web Services, the problem is to guarantee interoperability at all levels. We now look at semantics-based solutions to achieving interoperability: first, the Semantic Web Services efforts, second their application to middleware solutions, and third the database approaches.

4.2 Semantic Web Services

The problem of data interoperability is crucial to address the problem of service composition since, for two services to work together, they need to share a consistent interpretation of the data that they exchange. To this extent a number of efforts, which are generically known as Semantic Web Services, attempt to enrich the Web Services description languages with a description of the semantics of the data exchanged in the input and output messages of the operations performed by services. The result of these efforts are a set of languages that describe both the orchestration of the services' operations, in the sense of the possible sequences of messages that the services can exchange as well as the meanings of these messages with respect to some reference ontology.

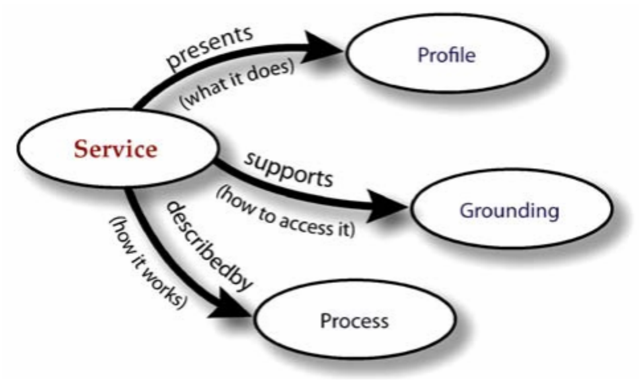


Fig. 8. OWL-S Upper Level Structure

OWL-S [26] and its predecessor DAML-S [25] have been the first efforts to exploit Semantic Web ontologies to enrich descriptions of services. The scope of OWL-S is quite broad, with the intention to support both service discovery through a representation of the capabilities of services, as well as service composition and invocation through a representation of the semantics of the operations and the messages of the service. As shown in Fig. 8, services in OWL-S are described at three different levels. The Profile describes the capabilities of the service in terms of the information transformation produced by the service, as well as the state transformation that the service produces; the Process (Model) that describes the workflow of the operations performed by the service, as well as the semantics of these operations, and the Grounding that grounds the abstract process descriptions to the concrete operation descriptions in WSDL.

In more detail, the information transformation described in the Profile is represented by the set of inputs that the service expects and outputs that it is expected to produce, while the state transformation is represented by a set of conditions (preconditions) that need to hold for the service to execute correctly and the results that follow the execution of the service. For example, a credit card registration service may produce an information transformation that takes personal information as input, and returns the issued credit card number as output; while the state transformation may list a number of (pre)conditions that the requester needs to satisfy, and produce the effect that the requester is issued the credit card corresponding to the number reported in output.

The Process Model and Grounding relate more closely to the invocation of the service and therefore address more directly the problem of data interoperability. The description of processes in OWL-S is quite complicated, but in a nutshell they represent a transformation very similar to the transformation described by the Profile in the sense that they have inputs, outputs, preconditions and results that describe the information transformation as well as the state transformation which results from the execution of the process. Furthermore, processes are divided into two categories: *atomic processes* that describe atomic actions that the service can perform, and *composite processes* that describe the workflow control structure.

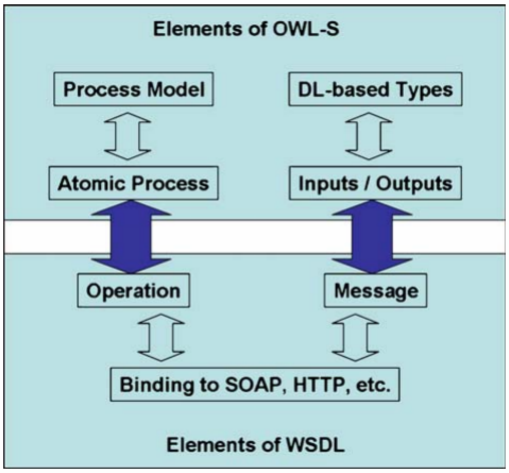


Fig. 9. The structure of the OWL-S process grounding

In turn atomic processes “ground” into WSDL operations as shown in Fig. 9 by mapping the abstract semantic descriptions of inputs and outputs of process into the WSDL message structures. In more detail, the grounding specifies which operations correspond to an atomic process, and how the abstract semantic representation is transformed in the input messages of the service or derived from the output messages. One important aspect of the Grounding is that it separates the OWL-S description of the service from the actual implementation of the service, and therefore, every service which can be expressed in WSDL, can be represented in OWL-S. As a result of the service description provided by OWL-S the client service would always know how to derive the message semantics from the input/output messages of the service. Ideally therefore, the client may represent its own information at the semantic level, and then ground it to into the messages exchanged by the services.

Analysis of OWL-S. OWL-S provides a mechanism for addressing the data semantics; however it has failed in a number of aspects. First, many aspects of the service representation are problematic; for example, it is not clear what is the relation between the data representation of the atomic processes and the input/output representation of the complex (control flow) processes. Second, OWL-S is limited to a strict client/server model, as supported by WSDL, as a consequence it is quite unclear how OWL-S can be used to derive interoperability connectors in other types of systems. Third, OWL-S assumes the existence of an ontology that is shared between the client and server; this pushes the interoperability problem one level up. Of course the next data interoperability question is ``what if there is not such a shared ontology?''

SA-WSDL. Semantic Web Services reached the standardization level with SA-WSDL [27], which defines a minimal semantic extension of WSDL. SA-WSDL builds on the WSDL distinction between the abstract description of the service, which includes the WSDL 2.0 attributes Element Declaration, Type Definition and Interface, and the concrete description that includes Binding and Service attributes which directly link to the protocol and the port of the service. The objective of SA-WSDL is to provide an annotation mechanism for abstract WSDL. To this extent it extends WSDL with new attributes:

1. *modelReference*, to specify the association between a WSDL or XML Schema component and a concept in some semantic model;
2. *liftingSchemaMapping* and *loweringSchemaMapping*, that are added to XML Schema element declarations and type definitions for specifying mappings between semantic data and XML.

The *modelReference* attribute has the goal of defining the semantic type of the WSDL attribute to which it applies; the lifting and lowering schema mappings have a role similar to the mappings in OWL-S since their goal is to map the abstract semantic to the concrete WSDL specification. For example, when applied to an input message, the model reference would provide the semantic type of the message, while the *loweringSchemaMapping* would describe how the ontological type is transformed into the input message.

A number of important design decisions were made with SA-WSDL to increase its applicability. First, rather than defining a language that spans across the different levels of the WS stack, the authors of SA-WSDL have limited their scope to augmenting WSDL, which considerably simplifies the task of providing a semantic representation of services (but also limits expressiveness). Specifically, there is no intention in SA-WSDL to support the orchestration of operations. Second, there is a deliberate lack of commitment to the use of OWL [28] as an ontology language or to any other particular semantic representation technology. Instead, SAWSDL provides a very general annotation mechanism that can be used to refer to any form of semantic markup. The annotation referents could be expressed in OWL, in UML, or in any other suitable language. Third, an attempt has been made to maximize the use of available XML technology from XML schema, to XML scripts, to XPath, with the attempt to lower the entrance barrier to early adopters.

Analysis of SA-WSDL. Despite these design decisions that seem to suggest a sharp distinction from OWL-S, SA-WSDL shares features with OWL-S' WSDL grounding. In particular, both approaches provide semantic annotation attributes for WSDL, which are meant to be used in similar ways. It is therefore natural to expect that SAWSDL may facilitate the specification of the Grounding of OWL-S Web Services, a proposal in this direction has been put forward in [29]. The apparent simplicity of the approach is somewhat deceiving. First, SA-WSDL requires a solution to the two main problems of the semantic representation of Web Services: namely the generation and exploitation of ontologies, and the mapping between the ontology and the XML data that is transmitted through the wire. Both processes are very time consuming. Second, there is no obligation what-so-ever to define a modelReference or a schemaMapping for any of the attributes of the abstract WSDL, with the awkward result that it is possible to define the modelReference of a message but not how such model maps to the message, therefore it is impossible to map the abstract input description to the message to send to the service, or given the message of the service to derive its semantic representation. Conversely, when schemaMapping is given, but not the modelReference, the mapping is know but not the expected semantics of the message, with the result that it is very difficult to reason on the type of data to send or to expect from a service.

Web Service Modelling Ontology (WSMO) aims at providing a comprehensive framework for the representation and execution of services based on semantic information. Indeed, WSMO has been defined in conjunction with WSML (Web Service Modelling Language) [30], which provides the formal language for service representation, and WSMX (Web Service Modelling eXecution environment) [31] which provides a reference implementation for WSMO. WSMO adopts a very different approach to the modelling of Web Services than OWL-S and in general the rest of the WS community. Whereas the Web Service Representation Framework concentrates on the support of the different operations that can be done with Web Services, namely discovery with the Service Profile as well as UDDI [32], composition with the Process Model as well as BPEL4WS [33] and WS-CDL [34], and invocation with the Service Grounding, WSDL or SA-WSDL, WSMO provides a general framework for the representation of services that can be utilized to support the operations listed above, but more generally to reason about services and interoperability. To this extent it identifies four core elements:

- *Web Services*: which are the computational entities that provide access to the services. In turn their description needs to specify their capabilities, interfaces and internal mechanisms.
- *Goals*: that model the user view in the Web Service usage process.
- *Ontologies* provide the terminology used to describe Web Services and Goals in a machine processable way that allow other components and applications to take actual meaning into account.
- *Mediators*: that handle interoperability problems between different WSMO elements. We envision mediators as the core concept to resolve incompatibilities on the data, process and protocol level.

What is striking about WSMO with respect to the rest of the WS efforts (semantic and not) is the representation of goals and mediators as “first class citizens”. Both goals and mediators are represented as ``by product'' by the rest of the WS community. Specifically, in other efforts the users' goals are never specified, rather they are manifested through the requests that are provided to a service registry such as UDDI or to a service composition engine; on the other side mediators are either a type of service and therefore indistinguishable from other services, or generated on the fly through service composition to deal with interoperability problems. Ontologies are also an interesting concept in WSMO, because WSMO does not limit itself to use existing ontology languages, as in the case of OWL-S that is closely tied to OWL, nor it is completely agnostic as in the case of SA-WSDL. Rather WSMO relies on WSML which defines a family of ontological languages which are distinguished by logic assumptions and expressivity constraints. The result is that some WSML sub-languages are consistent (to some degree) with OWL, while others are inconsistent with OWL and relate instead to the DL family of logics.

Despite these differences, the description of Web Services has strong relations to other Web Services efforts. In this direction, WSMO grounds on the SA-WSDL effort (indeed SA-WSDL has been strongly supported by the WSMO initiative). Furthermore, the capabilities of a Web Service are defined by the state and information transformation produced by the execution of the Web Service, as was the case in OWL-S. The Interface of a Web Service is defined by providing a specification of its choreography which defines how to communicate with the Web Service in order to use its functions; and by the orchestration that reveals how the functionality of the service is achieved by the cooperation of more elementary Web Service providers. Of particular interest to addressing interoperability problems, WSMO defines three types of mediators:

1. *Data Level Mediation* - mediation between heterogeneous data sources, they are mainly concerned with ontology integration.
2. *Protocol Level Mediation* - mediation between heterogeneous communication protocols, they relate to choreographies of Web Services that ought to interact.
3. *Process Level Mediation* - mediation between heterogeneous business processes; this is concerned with mismatch handling on the business logic level of Web Services and they relate to the orchestration of Web Services.

Analysis of WSMO. WSMO put a strong emphasis on mediation and, as discussed above, it defines mediation as a "first class" citizen. The problem with WSMO is that that the WSMO project proposed an execution semantics for mediators [31] [35] but so far no theory or algorithm on how to construct mediators automatically has been proposed by the project. Somehow, it is curious that mediation is one of the fundamental elements of the approach while choreography is left to a secondary role within the specification of service definitions. Essentially it moves service composition to a secondary role in the theory.

4.3 Semantic Middleware

A number of research efforts have investigated middleware that support semantic specification of services for pervasive computing. These solutions mainly focus on providing middleware functionalities enabling semantic service discovery and composition as surveyed hereafter. **The Task Computing project** [36] is an effort for ontology-based dynamic service composition in pervasive computing environments. It relies on the UPnP service discovery protocol, enriched with semantic service descriptions given in OWL-S. Each user of the pervasive environment carries a service composition tool on his/her device that discovers on the fly available services in the user's vicinity and suggests to the user a set of possible compositions of these services. The user may then select the right composition among the suggested ones.

IGPF (**Integrated Global Pervasive Computing Framework**) [37] introduces a semantic Web Services-based middleware for pervasive computing. This middleware builds on top of the semantic Web paradigm to share knowledge between the heterogeneous devices that populate pervasive environments. The idea behind this framework is that information about the pervasive environments (i.e., context information) is stored in knowledge bases on the Web. This allows different pervasive environments to be semantically connected and to seamlessly pass user information (e.g., files/contact information), which allows users to receive relevant services. Based on these knowledge bases, the middleware supports the dynamic composition of pervasive services modelled as Web Services. These composite services are then shared across various pervasive environments via the Web.

The Ebiquity group describes a **semantic service discovery and composition protocol for pervasive computing**. The service discovery protocol, called GSD (Group-based Service Discovery) [38], groups service advertisements using an ontology of service functionalities. In this protocol, service advertisements are broadcasted to the network and cached by the networked nodes. Then, service discovery requests are selectively forwarded to some nodes of the network using the group information propagated with service advertisements. Based on the GSD service discovery protocol, the authors define a service composition functionality for infrastructure-less mobile environments [39]. Composition requests are sent to one of the composition managers of the environment, which performs a distributed discovery of the required component services.

The combined work in [40] and [41] introduces an efficient, **semantic, QoS-aware service-oriented middleware for pervasive computing**. The authors propose a semantic service model to support interoperability between existing semantic but also plain syntactic service description languages. The model further supports formal specification of service conversations as finite state automata, which enables automated reasoning about service behaviour independently of the underlying conversation specification language. Moreover, the model supports the specification of service non-functional properties to meet the specific requirements of pervasive applications. The authors further propose an efficient semantic service registry. This registry supports a set of conformance relations for matching both syntactic and rich semantic service descriptions, including non-functional properties. Conformance relations evaluate the semantic distance between service descriptions and rate services with respect to their suitability for a specific client request, so that selection can be made among them. Additionally, the registry supports efficient reasoning on semantic service descriptions by semantically organizing such descriptions and minimizing recourse to ontology-based reasoning, which makes it applicable to highly interactive pervasive environments. Lastly, the authors propose flexible QoS-aware service composition towards the realization of user-centric tasks abstractly described on the user's handheld. Flexibility is enabled by a set of composition algorithms that may be run alternatively according to the current resource constraints of the user's device. These algorithms support integration of services with complex behaviours into tasks also specified with a complex behaviour; and this is done efficiently relying on efficient formal techniques. The algorithms further support the fulfilment of the QoS requirements of user tasks by aggregating the QoS provided by the composed networked services.

The above surveyed solutions are indicative of how ontologies have been integrated into middleware for describing semantics of services in pervasive environments. Semantics of services, users and the environment are put into semantic descriptions, matched for service discovery, and composed for achieving service compositions. Focus is mainly on functional properties, while non-functional ones have been less investigated. Then, efficiency is a key issue for the resource-constrained pervasive environments, as reasoning based on ontologies is costly in terms of computation.

4.4 Beyond Web Services: DB Federation

The problem of data interoperation is by no means restricted to Web Services and middleware, rather it has been looked at the DB community for a long time. In this context, the data problem has been widely studied by the DB community while addressing the task of DB federation. Despite of the importance of the information stored in DBs, because of the way DBs and organizations evolve, the information stored on different databases is often very difficult to integrate. In this context "Database federation is one approach to data integration in which middleware, consisting of a relational database management system, provides uniform access to a number of heterogeneous data sources" [42]. Federated Data sources have a lot in common with the heterogeneous systems to be connected. They need to federate autonomous databases which are autonomously maintained, therefore they need to support a high degree of heterogeneity both at the architectural level, in the sense that they should host different version of databases made by different vendors as well support data heterogeneity because different nodes may follow different data schema.

The standard solution to the problem of data interoperability is to provide **Table User Defined Functions** (T-UDF) [42] which reformat the data from one database and present it in a format that is consistent with the format of a different data-base. For example, if one database provides address book information, a programmer may define a T-UDF addressbook()which reformats the data in the appropriate way, and then retrieve the data by using the SQL command FROM TABLE addressbook() in the query. T-UDF hardly provides a solution to the problem of data interoperability since they require a programmer that reformats the data from one data-schema to another.

Since the definition of translation functions as the T-UDF functions above is a very expensive process a considerable effort has been put into learning the translation between data-base schemata. Examples of these translations are provided in [43] [44]. They exploit a combination of machine learning, statistical processing and natural language lexical semantics to "guess" how two data-base schemata correspond. In Section 5.4 similar tools for ontology matching are analyzed more in detail.

The results of these mapping processes are mappings between data schemata that are correct up to a degree of confidence. The user should then find a way to deal with the reduced confidence in the results. One proposal in this direction has been provided by **Trio** [45], a data-base management system that extends the traditional data model to include data accuracy and lineage. Within Trio it is possible to express queries of the sort "find all values of X with approximation with confidence greater than K".

The approaches above ignore the most important information that is required for data mapping namely the explicit annotation of data semantics. Above, we discussed T-UDT as a mechanism for data translation mappings, but the problem with any form of mapping is that it makes assumptions on the semantics of the schemata that it is mapping across. There is therefore neither guarantee that these mappings are correct [46] nor that they will generalize if and when the schemata are modified. The automatic mapping mechanisms above, try to circumvent the problem of explicit semantics by using learning inference. But they assume semantics in the form of background knowledge such as lexical semantics without any guarantee that the background knowledge is relevant for the specific transformation. Essentially, the lack of explicit semantics emerges as an error in the accuracy of the transformation.

The development of ontologies, in the sense of shared data structures, is an alternative to the methods produced above. Essentially, instead of mapping all schemata directly in a hardcoded way as suggested by the T-UDT methods or try to guess the relation between schemata as suggested by the learning mechanisms, schemata are mapped to a unique "global" schema, indeed an ontology, from which direct mappings are derived. In this model the ontology provides the reference semantic for all schemata. The advantage of this model is that the DB provider could in principle provide the mapping to the ontology possibly removing the misinterpretation problem.

There are a number of problems of this approach. First, the ontology should be expressive enough to express all information within all the schemata in the federated databases. This implicitly requires a mechanism for extensible ontologies since adding new databases may require an extension of the ontology. Second, the derivation of mapping rules is proven to have an NP worst case computational complexity [47].

4.5 Raising Interoperability One Level Up

The discussion about ontologies above immediately raises the question of whether and to what extent ontologies just push the interoperability problem somewhere else. Ultimately, what guarantees that the interoperability problems that we observe at the data structure level do not appear again at the ontology level? Suppose that different middlewares refer to different ontologies, how can they interoperate?

The ideal way to address this problem is to construct an *alignment ontology*, such as SUMO[5], which provide a way to relate concepts in the different ontologies. Essentially, the alignment ontology provides a mapping that translates one ontology into the other. Of course, the creation of alignment ontologies not only requires efforts, but more importantly, it requires a commitment so that the aligning ontology is consistent with all ontologies to be aligned.

Such alignment ontologies, when possible, are very difficult to build and very expensive. To address this problem, in the context of the semantic web there is a very active subfield that goes under the label of *Ontology Matching* [48][49] which develops algorithms and heuristics to infer the relation between concepts in different

[5] SUMO stands for: *Suggested Upper Merged Ontology*. It is available at: http://www.ontologyportal.org/

ontologies. The result of an ontology matcher is a set of relations between concepts in different ontologies, and a level of confidence that that these relations hold. For example, an ontology matcher may infer that the concept *Price* in one ontology is equivalent to *Cost* in another ontology with a confidence of 0.95. In a sense, the confidence value assigned by the ontology matcher is a measure of the quality of the relations specified.

Ontology matching provides a way to address the problem of using different ontologies without pushing the data interoperability problem somewhere else. But this solution comes at a cost of the confidence on the on the interoperability solution adopted and ultimately on the overall system.

5 Analysis

The results of the state of the art investigation in Sections 3 and 4 shows two important things; first, there is a clear disconnect between the main stream middleware work and the work on application, data, and semantic interoperability; second, none of the current solutions addresses all of the requirements of dynamic pervasive systems as highlighted in the interoperability barriers in Section 2.

With respect to the first problem, it is clear that two different communities evolved independently. The first one, addressing the problems of middleware, has made a great deal of progress toward middleware that support sophisticated discovery and interaction between services and components. The second one, addressing the problem of semantic interoperability between services, however, inflexibly assuming Web Services as the underlying middleware; or the problem of semantic interoperability between data intensive components such as databases. The section on semantic middleware shows that ultimately the two communities are coming together, but a great deal of work is still required to merge the richness of the work performed on both sides.

With respect to the second problem, namely addressing the interoperability barriers from Section 2 we pointed out that in such systems endpoints are required to spontaneously discover and interact with one another and therefore these three fundamental dimensions are used to evaluate the different solutions:

1. *Does the approach resolve (or attempt to resolve) differences between discovery protocols employed to advertise the heterogeneous systems? [Discovery column]*
2. *Does the approach resolve (or attempt to resolve) differences between interaction protocols employed to allow communication with a system? [Interaction column]*
3. *Does the approach resolve (or attempt to resolve) data differences between the heterogeneous systems? [Data column]*
4. *Does the approach resolve (or attempt to resolve) the differences in terms of application behaviour and operations? [Application column]*
5. *Does the approach resolve (or attempt to resolve) the differences in terms of non-functional properties of the heterogeneous system? [Non-functional column]*

Table 1. Evaluation summary of effectiveness of interoperability solutions against each of the interoperability barriers

SD = Discovery
I = Interaction
D= Data
A = Application
N=Non-functional

	SD	I	D	A	N	Transparency
CORBA		X				CORBA for all
Web Services		X				WSDL & SOAP for all
ReMMoC	X	X				Client-side middleware
UIC		X				Client-side middleware
WSIF		X				Client-side middleware
MDA		X				Platform Independent models
UniFrame		X				Platform Specific models
ESB		X				Bridge connector
MUSDAC	X					Connection to middleware
INDISS	X					Yes
uMiddle	X	X				Yes
OSDA	X					Yes
SeDiM	X				X	Yes
SATIN	X	X				Choice of SATIN for all
Jini		X				Choice of Jini for all
Semantic Middleware			X	X		Choice of same semantic middleware for all
Semantic Web Services		X	X	X		WSDL for all plus commitment on a semantic framework and ontologies

The summary of this evaluation is in Table 1 (an x indicates: resolves or attempts to). This shows that no solution attempts to resolve all five dimensions of interoperability. Those that concentrate on application and data e.g. Semantic Web Services rely upon a common standard (WSDL) and conformance by all parties to use this with semantic technologies. Hence, transparent interoperability between dynamically communicating parties cannot be guaranteed. Semantic Web Services have a very broad scope, including discovery interaction and data interoperability, but these provide only a primitive support and languages to express the data dimension in the context of middleware solutions.

The transparency column shows that only the transparent interoperability solutions achieve interoperability transparency between all parties (however only for a subset of the dimensions). The other entries show the extent to which the application endpoint (client, server, peer, etc.) sees the interoperability solution. ReMMoC, UIC and WSIF rely on clients building the applications on top of the interoperability middleware; the remainder rely on all parties in the distributed system committing to a particular middleware or approach.

6 Conclusions and Future Work

This chapter has investigated the problem of interoperability in the complex distributed systems of today, with the added complexity stemming from the extreme level heterogeneity encountered in such systems coupled with the increasing level of dynamism of such systems which results in the need for spontaneous communication. The chapter highlights the key barriers to interoperability coupled with a discussion of solutions to interoperability featuring the research in the middleware community and related research on semantic interoperability. The most striking aspect of this study is that, while both communities focus on key interoperability problems, research efforts have to a large extent been disjoint. The other striking feature is that, despite considerable research efforts into interoperability dating back to the early 1980s, this remains a poorly understood area and currently solutions simply do not meet the needs on the complex distributed systems of today, particularly in terms of the levels of heterogeneity and dynamism as mentioned above.

The CONNECT project, an initiative funded under the Future and Emerging Technologies programme within the ICT theme of the European Commission's Framework programme, is taking a novel approach to the study of interoperability in complex distributed systems, going back to basics, and taking input from a variety of sub-disciplines including the middleware and semantic web communities, but also looking at supportive areas such as formal semantics of distributed systems, learning and synthesis technologies and support for dependable distributed systems. We propose an approach that:

- places semantic understanding of concepts at the heart of achieving interoperability,
- seeks a dynamic approach to interoperability where appropriate infrastructure is generated on-the-fly for the current context (emergent middleware), and this involves enabling technologies such as learning and synthesis of run-time connectors,
- grounds itself in formal semantics enabling validation and verification to be carried out,
- addresses the dependability requirements of modern distributed systems, including meeting the associated non-functional requirements in highly heterogeneous environments,
- supports dynamism allowing currently deployed solutions to be constantly monitored and adapted to changing context.

The rest of the book unfolds this story in more detail with chapter 2 providing an overview of the Connect architecture and other chapters unfolding key enabling technologies behind this approach.

References

1. Bouquet, P., Stoermer, H., Niederee, C., Mana, A.: Entity Name System: The Backbone of an Open and Scalable Web of Data. In: Proceedings of the IEEE International Conference on Semantic Computing (ICSC 2008), pp. 554–561 (2008)

2. Van Steen, M., Tanenbaum, A.: Distributed Systems: Principles and Paradigms. Prentice-Hall, Englewood Cliffs (2001)
3. Object Management Group.: The common object request broker: Architecture and specification Version 2.0. OMG Technical Report (1995)
4. Microsoft Corporation.: Distributed Component Object Model (DCOM) Remote Protocol Specification, `http://msdn.microsoft.com/en-gb/library/cc201989%28PROT.10%29.aspx`
5. Srinivasan. R.: RPC: Remote Procedure Call Protocol Specification Version 2. Network Working Group RFC1831 (1995), `http://tools.ietf.org/html/rfc1831`
6. Microsoft Corporation.: Microsoft Message Queuing, `http://www.microsoft.com/windowsserver2003/technologies/msmq/`
7. Carzaniga, A., Rosenblum, D., Wolf, A.: Design and Evaluation of a Wide-Area Event Notification Service. ACM Transactions on Computer Systems 19(3), 332–383 (2001)
8. Gelernter, D.: Generative communication in Linda. ACM Transactions on Programming Language and Systems 7(1), 80–112 (1985)
9. Wyckoff, P., McLaughry, S., Lehman, T., Ford, D.: Tspaces. IBM Systems Journal 37(3), 454–474 (1998)
10. Davies, N., Friday, A., Wade, S., Blair, G.: L^{2}imbo: A Distributed Systems Platform for Mobile Computing. ACM Mobile Networks and Applications (MONET) 3(2), 143–156 (1998)
11. Murphy, A., Picco, G., Roman, G.: LIME: A Middleware for logical and Physical Mobility. In: 21st International Conference on Distributed Computing Systems (ICDCS-21), pp. 524–533 (2001)
12. Booth D., Haas, H., McCabe, F., Newcomer, E., Champion, M., Ferris, C., Orchard, D.: Web Services Architecture. W3C Working Group Note (2004), `http://www.w3.org/TR/ws-arch/`
13. Roman, M., Kon, F., Campbell, R.: Reflective Middleware: From Your Desk to Your Hand. IEEE Distributed Systems Online 2(5) (2001)
14. Kon, F., Román, M., Liu, P., Mao, J., Yamane, T., Magalhães, L., Campbell, R.: Monitoring, security, and dynamic configuration with the *dynamicTAO* reflective ORB. In: Coulson, G., Sventek, J. (eds.) Middleware 2000. LNCS, vol. 1795, pp. 121–143. Springer, Heidelberg (2000)
15. Grace, P., Blair, G., Samuel, S.: A Reflective Framework for Discovery and Interaction in Heterogeneous Mobile Environments. ACM SIGMOBILE Mobile Computing and Communications Review 9(1), 2–14 (2005)
16. Duftler, M., Mukhi, N., Slominski, S., Weerawarana, S.: Web Services Invocation Framework (WSIF). In: Proceedings of OOPSLA 2001 Workshop on Object Oriented Web Services, Tampa, Florida (2001)
17. Object Management Group.: COM/CORBA Interworking Specification Part A & B. OMG Technical Report orbos/97-09-07 (1997)
18. Bromberg, Y., Issarny, V.: INDISS: Interoperable Discovery System for Networked Services. In: Alonso, G. (ed.) Middleware 2005. LNCS, vol. 3790, pp. 164–183. Springer, Heidelberg (2005)
19. Nakazawa, J., Tokuda, H., Edwards, W., Ramachandran, U.: A Bridging Framework for Universal Interoperability in Pervasive Systems. In: Proceedings of 26th IEEE International Conference on Distributed Computing Systems (ICDCS 2006), Lisbon, Portuga, (2006)

20. Limam, N., Ziembicki, J., Ahmed, R., Iraqi, Y., Li, D., Boutaba, R., Cuervo, F.: OSDA: Open service discovery architecture for efficient cross-domain service provisioning. Computer Communications 30(3), 546–563 (2007)
21. Flores, C., Grace, P., Blair, G.: SeDiM: A Middleware Framework for Interoperable Service Discovery in Heterogeneous Networks. ACM Transactions on Autonomous and Adaptive Systems 6(1), article 6 (2011)
22. Zachariadis, S., Mascolo, C., Emmerich, W.: Satin: A Component Model for Mobile Self-Organisation. In: Meersman, R., Tari, Z. (eds.) OTM 2004. LNCS, vol. 3291, pp. 1303–1321. Springer, Heidelberg (2004)
23. Arnold, K., O'Sullivan, B., Scheifler, R., Waldo, J., Wollrath, A.: The Jini Specification. Addison Wesley, Reading (1999)
24. Hammer, J., McLeod, D.: An approach to resolving semantic heterogenity in a federation of autonomous, heterogeneous database systems. Int. J. Cooperative Inf. Syst 2(1), 51–83 (1993)
25. Burstein, M., Hobbs, J., Lassila, O., Martin, D., McDermott, D., McIlraith, S., Narayanan, S., Paolucci, M., Payne, T., Sycara, K.: DAML-S: Web service description for the semantic web. In: Horrocks, I., Hendler, J. (eds.) ISWC 2002. LNCS, vol. 2342, pp. 348–363. Springer, Heidelberg (2002)
26. Martin, D., Burstein, M., Mcdermott, D., Mcilraith, S., Paolucci, M., Sycara, K., Mcguinness, D., Sirin, E., Srinivasan, N.: Bringing Semantics to Web Services: The OWL-S Approach. In: Cardoso, J., Sheth, A.P. (eds.) SWSWPC 2004. LNCS, vol. 3387, pp. 26–42. Springer, Heidelberg (2005)
27. Farrell J., Lausen, H.: Semantic Annotations for WSDL and XML Schema. W3C Recommendation (2007), `http://www.w3.org/TR/sawsdl/`
28. McGuinness D., Harmelen, F.: OWL Web Ontology Language. W3C recommendation (2004), `http://www.w3.org/TR/owl-features/`
29. Martin, D., Paolucci, M., Wagner, M.: Bringing Semantic Annotations to Web Services: OWL-S from the SAWSDL Perspective. In: Aberer, K., Choi, K.-S., Noy, N., Allemang, D., Lee, K.-I., Nixon, L.J.B., Golbeck, J., Mika, P., Maynard, D., Mizoguchi, R., Schreiber, G., Cudré-Mauroux, P. (eds.) ASWC 2007 and ISWC 2007. LNCS, vol. 4825, pp. 340–352. Springer, Heidelberg (2007)
30. de Bruijn, J., Lausen, H., Krummenacher, R., Polleres, A., Predoiu, L.: The Web Service Modeling Language WSML (2005), `http://www.wsmo.org/TR/d16/d16.1/v0.21/`
31. Haller, A., Cimpian, E., Mocan, A., Oren, E., Bussler, C.: WSMX - a semantic service-oriented architecture. In: Proceedings of the International Conference on Web Services (ICWS 2005), Orlando, Florida, pp. 321–328 (2005)
32. OASIS: Univeral Description, Discovery and Integration of Web Services (2002), `http://www.uddi.org`
33. Jordan D., Evdemon, J.: Web Services Business Process Execution Language (WSBPEL) Version 2.0. (2007), `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html`
34. Kavantzas N., Burdett, D., Ritzinger, G., Fletcher, T., Lafon, Y., Barreto, C.: Web Services Choreography Description Language Version 1.0. (2005), `http://www.w3.org/TR/ws-cdl-10/`
35. Cimpian, E., Mocan, A.: WSMX Process Mediation Based on Choreographies. In: Bussler, C.J., Haller, A. (eds.) BPM 2005. LNCS, vol. 3812, pp. 130–143. Springer, Heidelberg (2006)

36. Masuoka, R., Parsia, B., Labrou, Y.: Task Computing – the Semantic Web Meets Pervasive Computing. In: Fensel, D., Sycara, K., Mylopoulos, J. (eds.) ISWC 2003. LNCS, vol. 2870, pp. 866–881. Springer, Heidelberg (2003)
37. Singh, S., Puradkar, S., Lee, Y.: Ubiquitous Computing: Connecting Pervasive Computing Through Semantic Web. Information Systems and e-Business Management Journal 4(4), 421–439 (2005)
38. Chakraborty, D., Joshi, A., Finin, T.: Toward Distributed Service Discovery in Pervasive Computing Environments. IEEE Transactions on Mobile Computing 5(2), 97–112 (2006)
39. Chakraborty, D., Joshi, A., Finin, T., Yesha, Y.: Service Composition for Mobile Environments. Journal on Mobile Networking and Applications, Special Issue on Mobile Services 10(4), 435–451 (2005)
40. Ben Mokhtar, S., Georgantas, N., Issarny, V.: COCOA: COnversation-based Service Composition in PervAsive Computing Environments with QoS Support. Journal of Systems and Software, Special Issue on ICPS 2006 80(12), 1941–1955 (2007)
41. Ben Mokhtar, S., Preuveneers, D., Georgantas, N., Issarny, V., Berbers, Y.: EASY: Efficient SemAntic Service DiscoverY in Pervasive Computing Environments with QoS and Context Support. Journal of Systems and Software, Special Issue on Web Services Modelling and Testing 81(5), 785–808 (2008)
42. Haas, M., Lin, E., Roth, M.: Data integration through database federation. IBM Systems Journal 41(4), 578–596 (2002)
43. Jung, J.: Taxonomy alignment for interoperability between heterogeneous virtual organizations. Expert Systems with Applications 34(4), 2721–2731 (2008)
44. Berlin, J., Motro, A.: Database schema matching using machine learning with feature selection. In: Pidduck, A.B., Mylopoulos, J., Woo, C.C., Ozsu, M.T. (eds.) CAiSE 2002. LNCS, vol. 2348, pp. 452–466. Springer, Heidelberg (2002)
45. Widom, J.: Trio: A System for Integrated Management of Data, Accuracy, and Lineage. In: Second Biennial Conference on Innovative Data Systems Research (CIDR 2005), Pacific Grove, California (2005)
46. Vetere, G., Lenzerini, M.: Models for semantic interoperability in service-oriented architectures. IBM Systems Journal 44(4), 887–904 (2005)
47. Fagin, P., Kolaitis, P., Popa, L.: Data Exchange, Getting to the Core. In: Symposium of Principles of Database Systems, pp. 90–101. ACM, New York (2003)
48. Euzena, J., Shvaiko, P.: Ontology matching. Springer, Heidelberg (2007)
49. Shvaiko, P., Euzenat J., Giunchiglia F., Stuckenschmidt H., Mao, M. Cruz, I.: Proceedings of the 5th International Workshop on Ontology Matching (OM 2010). CEUR (2010)

The CONNECT Architecture

Paul Grace[1], Nikolaos Georgantas[2], Amel Bennaceur[2], Gordon S. Blair[1], Franck Chauvel[3], Valérie Issarny[2], Massimo Paolucci[4], Rachid Saadi[2], Betrand Souville[4], and Daniel Sykes[2]

[1] School of Computing and Communications, Lancaster University, UK
p.grace@lancaster.ac.uk
[2] INRIA, CRI Paris-Rocquencourt, France
{nikolaos.georgantas,amel.bennaceur,valerie.issarny, rachid.saadi,daniel.sykes}@inria.fr
[3] School of Electronics Engineering and Computer Science, Peking University, China
franck.chauvel@sei.pku.edu.cn
[4] Laboratories Europe GmbH, Munich, Germany
{paolucci,souville}@docomolab-euro.com

Abstract. Current solutions to interoperability remain limited with respect to highly dynamic and heterogeneous environments, where systems encounter one another spontaneously. In this chapter, we introduce the CONNECT architecture, which puts forward a fundamentally different method to tackle the interoperability problem. The philosophy is to observe networked systems in action, learn their behaviour and then dynamically generate mediator software which will connect two heterogeneous systems. We present a high-level overview of how CONNECT operates in practice and subsequently provide a simple example to illustrate the architecture in action.

Keywords: Interoperability, emergent middleware, modelling, synthesis, middleware, protocol.

1 Introduction

1.1 Motivation: The Interoperability Problem

Interoperability is a measure of the ability of systems to connect,understand and exchange data with one another. As such, it reveals one of the fundamental problems in computer science. Indeed, the world wide budget for Interoperability is estimated to be in excess of $1 Trillion [7]). In the chapter *'Interoperability in Complex Distributed Systems'* of this book that surveys the interoperability problems and state of the art solutions [5], the important barriers to fully achieving interoperability are identified as:

– *Data heterogeneity.* Applications may use data that is represented in different ways and/or have different meanings.

- *Middleware heterogeneity.* Different protocols are used to advertise and search for services, e.g., Service Location Protocol (SLP), Jini, Universal Plug and Play (UPnP), and Lightweight Directory Access Protocol (LDAP). Further, services use different protocols to exchange and use data, e.g., Remote Method Invocation protocols such as SOAP, Java RMI and IIOP; or different messaging protocols such as Java Message Service (JMS) or Microsoft Message Queuing (MSMQ).
- *Application heterogeneity.* The application interfaces may be different in terms of the descriptions of operations, e.g., the behaviour provided by one operation in one interface may be provided by multiple operations in the other interface. Interfaces may also be heterogeneous in terms of the order in which operations must/should be called.
- *Heterogeneity of non-functional properties.* Systems may have particular non-functional properties, e.g., latency of message delivery, dependability measures and security requirements that must be resolved with respect to the connected system.

As a traditional solution to this problem, middleware-based standards (e.g. Web Services [6] or CORBA [26]) allow systems to be designed in advance in order to interoperate with each other. However, where environments are heterogeneous and dynamic (e.g. pervasive computing) such standards cannot be agreed upon in advance, nor can they deal with the heterogeneity of the networked systems in these environments. Interoperability platforms and transparent interoperability solutions offer more dynamic approaches. Interoperability platforms such as ReMMoC [16] and UIC [30], allow clients to be developed transparently from the heterogeneous middleware that may be spontaneously encountered in the future; these plug-in software at runtime that can communicate with the encountered protocol. While suitable for systems that know they will need to interoperate with heterogeneous protocol, this approach cannot this cannot solve the problem of two legacy platforms required to interoperate with one another; INDISS [8] and uMiddle [24] are examples of transparent interoperability solutions that dynamically translate through an intermediary language to achieve this requirement. However, in all of these cases, only a subset of the above four barriers are attempted to be resolved; see [5] for a detailed analysis of the state of the art which illustrates this observation.

Therefore, we advocate that new approaches are required to tackle interoperability in a fundamentally different way to achieve the objective of *universal and long-lived interoperability.* This goal is akin to the ideas of *universal translation*, a common device often appearing in science fiction; for example, the Babel Fish in "The Hitchhikers Guide to the Galaxy" [1] offers universal translation to allow native speech to be automatically and transparently translated to the language of any of the listeners, i.e., everyone speaks and hears their own language.

1.2 The CONNECT Approach

The approach of CONNECT is to produce *emergent middleware*, i.e., rather than create another middleware technology that is destined to be yet another legacy platform that in turn adds to the interoperability problem, we propose the novel approach of *generating the required middleware at runtime*, i.e., we synthesize the necessary software to connect (translate between) two end-systems. For example, if a client application developed using SOAP [17] encounters a CORBA server then the framework generates a Connector that resolves the heterogeneity of the data exchanged, the application behaviour, and the lower level middleware and network communication protocols.

To underpin the creation of *emergent middleware*, the Connect architecture performs the following important phases of system behaviour.

- *Discovering the functionality* of networked systems and applications advertised by legacy discovery protocols, e.g., Service Location Protocol (SLP) and Simple Service Discovery Protocol (SSDP). Then, transforming this discovered information to a rich intermediary description (the Connect Networked System Model) that can then be used to syntactically and semantically match heterogeneous services.
- *Using learning algorithms to dynamically determine the interaction behaviour* of a networked system from its intermediary representation and producing a model of this behaviour in the form of a labelled transition system (LTS). A full description of how learning is enabled in Connect is provided in [19].
- *Dynamically synthesizing* a software mediator. Taking as input the Networked System model and the learned LTS of two networked systems, Connect uses a formal approach to match the application behaviour of these systems and then map them onto one another to form the application mediator (to resolve the application behaviour differences); more detailed information about this method is provided in the chapter 'Application-layer Connector Synthesis' [28]. Further, the differences in the middleware protocols are resolved through a similar formal method for matching and mapping of middleware protocols to produce middleware mediation methods; this is presented in the chapter 'Middleware-layer Connector Synthesis' [20]. The combination of the synthesized application-level and middleware-level mediators form the Connector mediator.
- *Deployment in the network environment.* The Connector mediator is made concrete by deploying it upon appropriate *listeners* and *actuators* that can communicate directly with networked systems using their legacy protocols.
- *Verification & validation* of the Connector is performed by enablers during mediator synthesis phase and also after deployment to ensure the correctness of the Connector and the running Connected system with respect to the requirements (and importantly the non-functional requirements) and intents of the involved networked systems. This process ensures the long-lived nature of a Connect solution. The methods to perform verification and validation are provided in the chapter 'Dependability and Performance Assessment of Dynamic Connected Systems' [3]

1.3 Structure of the Chapter

This chapter first provides a broad overview of the Connect architecture, identifying the key functions and principles, and then a simple example is utilised to illustrate how the overall architecture operates. Only a subset of the technical details are introduced, instead the chapter points the interested reader to other publications (including further chapters of this book) in order to discover the richer details and formal methods. The chapter is organised as follows. The overall Connect architecture is presented in Section 2; in particular this highlights how networked systems are first discovered and modelled, and then how the emergent CONNECTors between them are realised. To illustrate an important feature of the architecture, a description of the technologies employed to deploy CONNECTors is given in Section 3, here the methods to dynamically generate middleware protocol listeners and actuators are discussed. We then present a case study showing how a CORBA-based networked system achieves interoperability with a SOAP asynchronous messaging networked system using the Connect architecture in Section 4. Finally, in Section 5 we offer conclusions about the architecture and then pinpoint areas of interest for future research.

2 A Framework for Interoperability

2.1 CONNECT Actors

Before exploring the details of the Connect architecture we first introduce the key actors that are involved in the Connect process. These are central to the underlying architectural principles:

- *Networked systems* are systems that manifest the will to connect to other systems for fulfilling some intent identified by their users and the applications executing upon them.
- *Enablers* are networked entities in the environment of networked systems that incorporate all the intelligence and logic offered by Connect for enabling connection between heterogeneous networked systems. Enablers constitute the Connect enabling architecture.
- CONNECTors are the emergent connectors produced by the action of enablers.
- CONNECTed systems are the outcome of the successful creation and deployment of CONNECTors.

A high-level view of these actors is shown in Figure 1. It can be seen that networked systems manifest their will to connect. This will, along with information about the networked systems, is communicated in the form of some input to the enablers. One or more enablers collaborate to synthesize and deploy a CONNECTor that enables networked systems to connect and fulfill their individual intents.

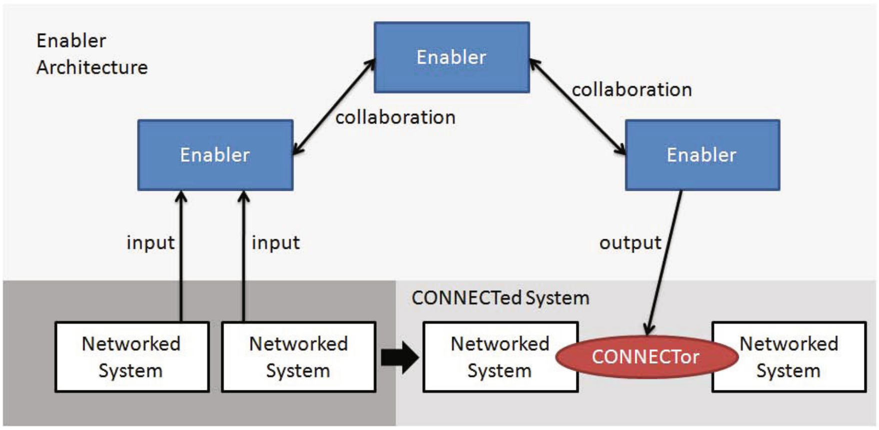


Fig. 1. Actors in the CONNECT architecture

2.2 Networked System Model

CONNECT seeks to observe, learn and model the external interaction behaviour of a networked system. This model, termed the *Networked System Model* is central to the CONNECT architecture and contains the information required by the enablers to produce the CONNECTors that ensure heterogeneous networked systems interoperate. There are two levels of interaction that must be considered by the model:

– *Middleware-layer interaction.* This includes information about the interaction protocol and the underlying network transport: what are the messages, their data content and format, and their sequence? The middleware semantics will also be covered, i.e., is this client-server, peer-to-peer, etc? Is the communication paradigm message-based or event-based, synchronous or asynchronous?
– *Application-layer interaction.* The application component describes: an intent, what external behaviour it requires, and what external behaviour it provides. The essential feature here is the interface, that is, a description of the set of functionalities of the component made accessible to (but also required from) its environment. Typically, this description comes in the form of a set of data inputs and associated outputs following a specific data type system. The application-layer will also describe its behaviour in terms of the sequence of application operations, and also the associated non-functional requirements of this behaviour.

The CONNECT Networked System model takes these abstract elements that are typically spread across different service descriptions, and the corresponding languages (e.g. Interface descriptions in WSDL [10], semantic annotations in SA-WSDL [12], and behaviour in BPEL), and integrates them into a uniform model

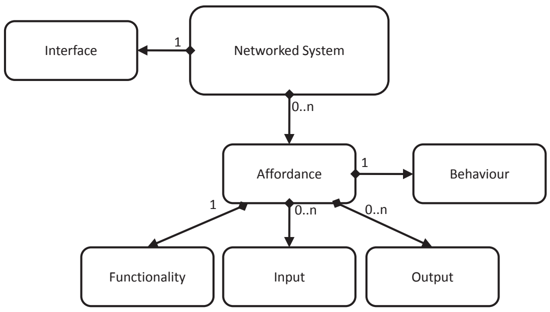


Fig. 2. Overview of the Networked System Model

that can be shared, understood and processed by the enablers. A high-level overview of this model is shown in Figure 2 and importantly highlights the key features of the model:

– The *affordance* is a macroscopic view, or the quality of a feature, of a networked system. Essentially the affordance describes the high-level roles a networked system plays, e.g., 'prints a document', or 'sends an e-mail'. This allows semantically equivalent action-relationships/interactions with another networked system to be matched; in short, they are complementarily providing/requesting the same thing.
– *Interfaces* provide a refined or a microscopic view of the system by specifying finer actions or methods that can be performed by/on the networked system, and used to implement its affordances. Each networked system is associated with a unique interface. The non-functional requirements of the interface operations are also described.
– The *behaviour description* documents the application behaviour in terms of how the actions of the interface are co-ordinated to achieve the system's affordance, and in particular how these are related to the underlying middleware functions. A BPEL-based specification language is employed to specify this behaviour.

2.3 The CONNECT Enabler Architecture

As previously identified, it is the Connect enablers whose role is to co-ordinate in order to produce a Connector that will ensure two legacy applications can interact. These enablers follow an important sequence of behaviour that we now identify:

– *Discovery* enables networked systems to manifest their will to connect to other networked systems and to discover mutually interested networked systems, while at the same time allows the Connect enabling architecture to retrieve initial information on likely-to-be-associated networked systems.

- *Learning* is performed by enablers upon networked systems for completing the initial information about the latter provided by discovery. The outcome of combined discovery and learning should be a sufficiently good Networked System Model of a networked system.
- *Synthesis & deployment* is performed by enablers for generating and deploying an appropriate CONNECTor that will successfully bridge the heterogeneous systems and establish a CONNECTed system.
- *Verification & validation* is performed by enablers during and after the synthesis phase for ensuring the correctness of the CONNECTor and the running CONNECTed system with respect to the requirements and intents of the involved networked systems.

These phases of behaviour are then split into software components each responsible for a particular role; hence this software component becomes a CONNECT enabler. The Enabler architecture is then the configuration of these enabler components which are deployed in the network environment and remotely communicate with each other. Figure 3 illustrates how these combine to achieve the particular goal of CONNECT, i.e., to take two networked systems whose heterogeneity denies them from interoperating with one another, learn their behaviour, identify a solution to ensure they interoperate, and then synthesize and deploy the required CONNECTor. We discuss the individual enablers in turn and describe how they communicate.

The Discovery Enabler. The discovery enabler leverages existing service discovery protocols such as SLP [18], UPnP [15], and WS-Discovery [25] in order to initially find out what networked systems are operating in the environment, what their intent and requirements for connection are, and whether other networked systems match these requirements. The discovery enabler receives both the advertisement messages and lookup request messages that are sent within the network environment by listening on known multicast addresses (used by legacy discovery protocols). These messages are then processed and their information from the legacy messages is extracted to form a *partial networked system model* for each of the networked systems, where the partial model consists of the affordance, and the application interface as shown in Figure 2. Further the discovery enabler can also extract information about the middleware protocols employed to provide initial input to the model of behaviour in Figure 2; for example, this information could be extracted from the WSDL binding element [10] in the case of WS-Discovery, or by parsing the protocol part of the URL returned by a discovery protocol (as in th case of SLP [18] and Bonjour[1]).

Initial matching is performed between discovered systems to determine whether two networked systems are candidates to have a CONNECTor generated between. The matching method examines the affordances of the two systems and employs ontology-based matching to identify if the two are a good match. On a match, the CONNECT process is initiated; first the current partial Networked System Model of each system is sent to the *learning enabler*, which then adds to the

[1] http://developer.apple.com/networking/bonjour/specs.html

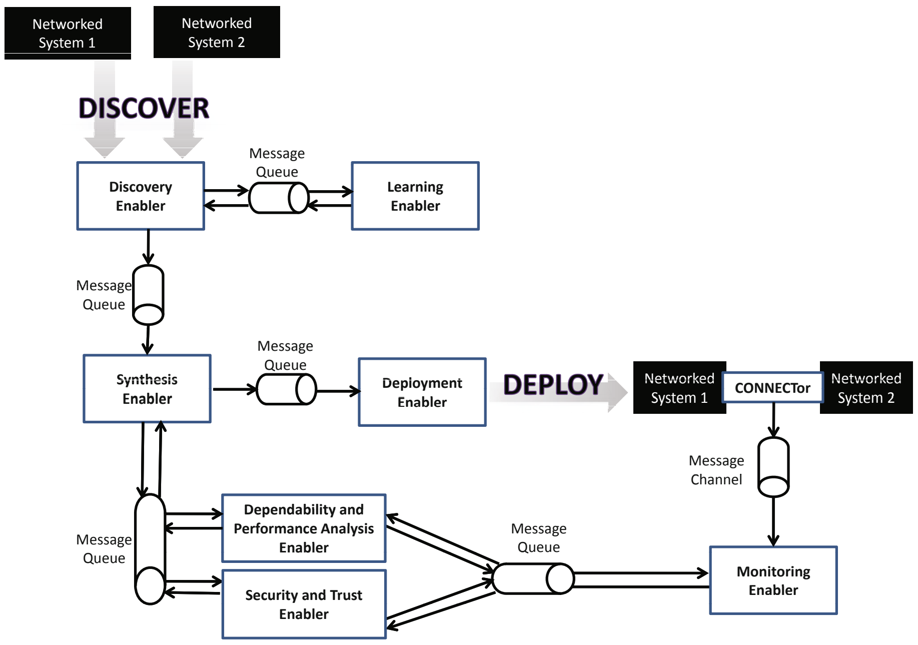


Fig. 3. The CONNECT Enabler architecture

behaviour description to the model to complete a richer view of the system's behaviour. On the completion of the Networked System Model, the discovery enabler sends this model to the *synthesis enabler*.

The Learning Enabler. The learning enabler uses active learning algorithms to dynamically determine the interaction behaviour of a networked system from its intermediary representation and produces a model of this behaviour in the form of a Labeled Transition System (LTS); this employs methods based on monitoring and model-based testing of the networked systems to elicit their interaction behaviour. The implementation of the enabler is built upon the LearnLib tool [29]. The learning method utilises two inputs: i) the interface description in the Networked System Model, and ii) the semantic annotations (that annotate the interface) which provide richer meanings to the tool. The learning enabler produces an LTS describing the interaction behaviour; this added to the behaviour section of the Networked System Model, and the outcome is a complete - as far as possible - instantiated networked system model. This is sent back to the discovery enabler to complete the discovery of the description of networked systems.

Synthesis Enabler. The role of the synthesis enabler is to take the Networked System Models of two systems and then synthesize the mediator component that is employed by the CONNECTor to co-ordinate the interaction between the two. Here, the synthesis enabler creates a mediator to resolve: i) application-level interoperability, and ii) middleware level interoperability. The LTS received from the discovery and learning phase is middleware specific, i.e., the transitions are strongly correlated to the behaviour of the middleware protocol. The first step is to abstract the behaviour of the system in a middleware-agnostic way to capture the application behaviour. This mapping is underpinned by a set of middleware rules and domain ontologies that describe how middleware behaviour can be abstracted towards a common representation of application behaviour– the *middleware agnostic LTS*). The methods to create middleware agnostic LTS are described in [20].

The next step is to create a common (application-level) abstraction of the networked systems. This method takes into account the ontology-based specification of each networked system and the common ontology specification for the application domain to produce corresponding abstract LTS for the middleware-agnostic LTS. Once complete, the two LTS can be matched and mapped to create the mediator. First, the existence of common traces in the LTS that lead the two systems to achieve a common goal is automatically checked; if there is a match and at least one common trace is found (which leads to achieve the specified common goal), the mapping between the two LTSs, over the common traces, is automatically performed and producing an abstract LTS that models the interaction behavior of the mediator. This is only a brief overview of this method and further information can be found in [28].;

Finally, the abstract LTS is made concrete by reapplying the middleware-specific information that was abstracted upon earlier in the method; this produces the concrete CONNECTor LTS that can be synthesized to create the software that can be directly deployed in the CONNECTors between the two legacy networked systems. From this the software The synthesis enabler can then output two alternative software types (depending upon the style of CONNECTor in use):

- *Mediator code.* The synthesis enabler generates the Java executable code that can be deployed directly as part of a CONNECTor configuration.
- An *'executable' LTS model.* The concrete LTS model can be sent directly, in order for it to be used by the mediation engine of a CONNECTor.

Either of these two outputs is sent to the deployment enabler in order to complete the construction of the CONNECTor.

Deployment Enabler. The Deployment Enabler receives as input the mediator code (or the LTS model) and the original Networked System Models; its objective is to finalise and then deploy the CONNECTor in each case. In order to do this, the enabler executes two important roles:

- It composes the required functionality to ensure that CONNECTors will communicate with the legacy networked systems, i.e., it will add the listeners and actuators to the mediator generated by the Synthesis Enabler. We discuss how the listeners and actuators are realised in Section 3.

– It deploys and manages the executable code (or the LTS model) of the Connectors in the network. For this, the enabler utilises OSGi[2] techniques; that is, the components that form the Connectors are bundled into OSGi components this allows them to be automatically deployed and executed upon network hosts running an OSGi platform (after being downloaded to the appropriate location); that is, the components that form the Connectors are bundled into OSGi components this allows them to be automatically deployed and executed upon network hosts running an OSGi platform (after being downloaded to the appropriate location).

Dependability and Performance Analysis/Security and Trust (SXT) Enabler. Once a Connector specification has been produced by the synthesis enabler it sends it to the dependability and performance analysis enabler to determine if the non-functional requirements (as described in the Networked System Model of each networked system) are satisfied. If so, the enabler tells the synthesis enabler to go ahead and deploy; otherwise, the dependability enabler enhances the initial LTS in order that it better meets the requirements of the connection; these enhanced models are returned to the synthesis enabler. The dependability enabler also continuously determines if the Connector maintains its non-functional requirements (as identified in the networked system's interface). It receives monitoring data from the monitoring enabler and in the case where there is no longer compliance, the dependability enabler sends a new specification to the synthesis enabler to initiate redeployment of a suitable Connector in the current conditions.

Monitoring Enabler. The monitoring enabler receives requests concerning which Connectors to monitor and then collects raw information about the Connectors by monitoring data that this Connector publishes to the monitoring channel. The derived data is passed to the dependability enabler to determine if the original non-functional requirements are being matched.

The Connect Message Bus. The enablers and Connectors use a simple message-based communication model to exchange information with one another. A Java Messaging Service (JMS) implementation[3] is used to implement the Message Bus. The reason for this choice of communication model is that two styles of communication are important in the Connect enabler architecture and are both provided by the technology:

– Point-to-Point exchange between enablers. As described earlier, the enablers send content (e.g., models and code) to be processed by a specific party, e.g., the discovery and learning enabler communicating to build the Networked System Model. JMS allows the behaviour to be achieved using a message queue as illustrated in Figure 3.

[2] http://www.osgi.org
[3] http://www.oracle.com/technetwork/java/index-jsp-142945.html

– Publish-Subscribe communication regarding Connector behaviour. The Connectors produce events in order for them to be monitored; enablers can subscribe to the channels that the Connectors publish these events to. For example, in Figure 3 the monitoring enabler subscribes to this channel in order to monitor Connector events.

2.4 CONNECTors

We now introduce the software elements that make up an individual Connector and also how they interact in order to achieve interoperability. This Connector architecture is illustrated in Figure 4. The software elements are described as follows:

– A *Listener* receives network messages (from the network engine) in the form of data packets and parses them according to the message format employed by the protocol that this message is specified by. Hence, each Listener parses messages from a single protocol, e.g., the SOAP listener parses SOAP messages. A listener produces an *Abstract Message* (see Section 3 for more information about abstract messages) that contains the information found in the original data packet, providing a uniform representation that can be manipulated and understood by the other elements in the Connector architecture. The API of the listener in Java is shown in Figure 5, the packet in a byte array is passed to the `MessageParse` method and a Java Object (`AbstractMessage`) representing the Abstract Message is produced.
– An *Actuator* performs the reverse role of a listener, i.e., it composes network messages according to a given middleware protocol, e.g., the SOAP Actuator creates SOAP messages. Actuators receive the Abstract Message and translate this into the data packet to be sent on the network via the network engine. The API of the actuator in Java is shown in Figure 5, a byte array is produced when the `AbstractMessage` object is passed to the `MessageCompose` method.

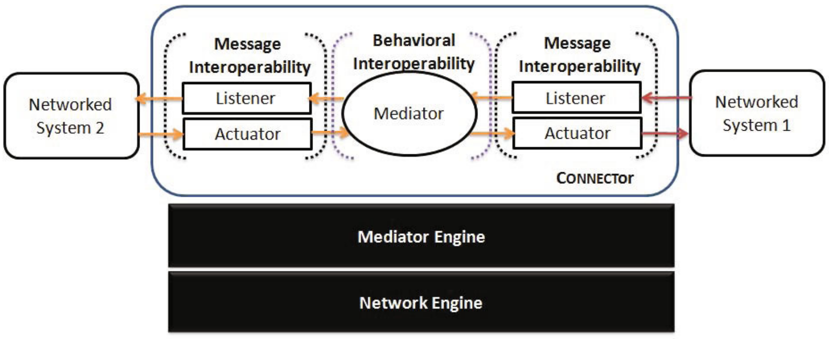
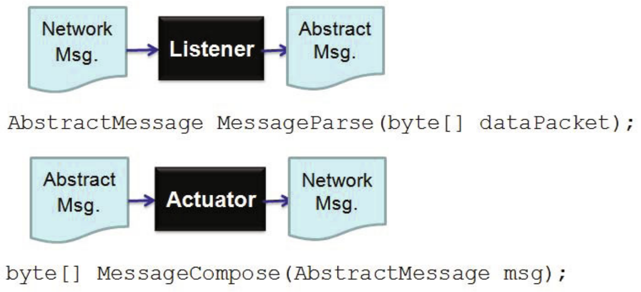


Fig. 4. The Connector Architecture

- The *Mediator* forms the central co-ordination element of a generated Connector. Its role is to translate the content received from one protocol (using `Abstract Message`) into the content required to send to the corresponding protocol. The mediator therefore addresses the the challenges of mapping between: different message content and formats, and different protocol behaviour, e.g., sequence of messages.
- The *Network Engine* provides a library of transport protocols with a common uniform interface to send and receive messages. Hence, it is possible to receive messages and send messages from multicast (e.g. IP multicast), broadcast and unicast transport protocols (e.g. UDP and TCP). The uniform interface provided by the network engine is similar to network programming libraries provided.
- The *Mediator engine* in the figure is an optional element of the architecture depending upon the implementation approach taken for mediators. The behaviour of the mediator is determined by a high-level model determining the operations to take. In the case where this model is turned directly into code there is no need for a mediation engine. In the case where the mediator model is an executable model (e.g., a BPEL specification, or an alternative Connect mediator model) then it is the mediation engine which executes these scripts. This flexibility in the intermediary architecture allows us to investigate the benefits of the two approaches, i.e., to investigate the performance gains of direct code generation, versus the ability to easily adapt the behaviour of the Connector at runtime when it is a model executed on the mediation engine..



```
AbstractMessage MessageParse(byte[] dataPacket);
```

```
byte[] MessageCompose(AbstractMessage msg);
```

Fig. 5. Listeners and Actuators API

2.5 Summary

This section has introduced the overall Connect architecture that puts the philosophy of discovery, learning and synthesis of Connectors into practice. Further information about the behaviour of Connect enablers can be found in chapters of this book [3] [20] [28]. We will now look more closely at the problem of communicating with networked systems, i.e., how the software mediators can send and receive messages in the protocols that are utilised. For example, if the networked systems use SOAP and IIOP how can the mediator send and receive SOAP and IIOP messages.

3 Communicating with Legacy Protocols

Connectors work by taking the concrete messages of legacy protocols and then creating an abstract representation of this data (the abstract message) such that it can be used to translate to one or more messages of a different legacy protocol. The translated abstract message then being composed into the concrete message format of the destination protocol. To illustrate this, consider Figure 6 which shows two protocol messages broken down into their field content; the message on the left is an SLP lookup message, whereas the message on the right is an SSDP lookup message. Both are performing the same function searching for a service of a given service type (this is the data contained in the `SrvType string` field of SLP and the `Service Target` field of SSDP). To achieve interoperability between them we need to extract data from the original concrete message, translate this, and then compose new concrete messages. This is a key underlying principle of the Connect architecture and in this section we discuss techniques to manipulate network messages. We first introduce the concept of abstract message, and then present solutions to marshall and unmarshall legacy protocol messages to/from this representation.

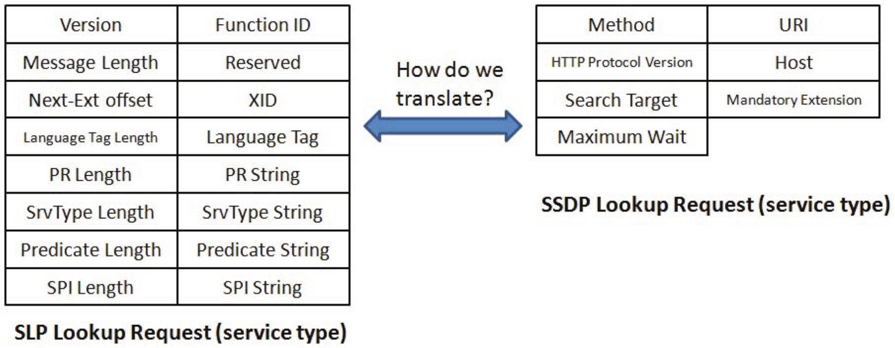


Fig. 6. Message formats of heterogeneous protocols–SLP and SSDP

3.1 Abstract Messages

A network message (as employed by a legacy communication protocol) is typically organized as a sequence of text lines for text-based protocols, or of bits, for a binary protocol. Messages are composed of fields. A Connector must extract relevant fields from the received message and use them to create one or more messages according to the target protocols. Similarly, it must extract relevant fields from the received responses and ultimately create a response according to the source protocol. Hence, the design of CONNECTors is based upon these message-based events; and the key design principle is to derive information from network messages and then describe them in a protocol independent manner. We term this protocol independent description of a message: **the *Abstract Message***. Received network messages are converted to an Abstract Message, correspondingly the Abstract Message is used to build the network message that must be sent.

```
<xsd:schema>
 <xsd:element name="Field">
   <xsd:complexType>
      <xsd:sequence>
         <xsd:element name="label" type="xsd:string"/>
         <xsd:element name="length" type="xsd:integer"/>
         <xsd:element name="type" type="xsd:string"/>
         <xsd:element name="mandatory" type="xsd:boolean"/>
         <xsd:element name="value" type="xsd:any"/>
         <xsd:element ref="Field" minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
   </xsd:complexType>
 </xsd:element>

 <xsd:element name="AbstractMessage">
   <xsd:complexType>
       <xsd:sequence>
         <xsd:element name="Name" type="xsd:string"/>
         <xsd:element ref="Field" minOccurs="0" maxOccurs="unbounded"/>
       </xsd:sequence>
   </xsd:complexType>
 </xsd:element>
</xsd:schema>
```

Fig. 7. The Abstract Message Schema

The schema for the Abstract Message content is illustrated in Figure 7. This shows that an Abstract Message consists of a set of fields; a field can be either primitive or structured. A *primitive field* is composed of a label naming the field, a type describing the type of the data content, a length defining the length in bits of the field, a boolean stating if this is a mandatory or optional field, and the value of the field, i.e., the data content. A *structured field* is composed of multiple primitive fields. For example, a URL field is composed of four primitive fields: the protocol, the address, the port, and the resource location.

Abstract Messages then represent the interface between the Listeners, Actuators and the Mediator, and the underlying network messages themselves. In order to achieve interoperability dynamically, the CONNECTor receives network messages from a networked system (in the format of the protocol employed by this legacy system). This event will trigger the execution of the Mediator, whose behaviour will determine the sequence of actions that manipulate the listeners and actuators. For example, it may receive one or more messages in the Abstract Message format and it may send one or more messages by composing a new Abstract Message and sending this to an Actuator to be delivered to the target networked system.

3.2 From Abstract Message to Concrete Message

To form a CONNECTor the mediator must be able to communicate with the networked systems using their legacy protocol. Hence, the mediator is composed with Listeners and Actuators as described earlier in the vision of the CONNECT architecture (see Section 2.4). Within CONNECT, the general philosophy employed for the deployment of Listeners and Actuators is to utilise DSLs to describe protocol messages. These high-level descriptions are then used to create the software components that will be deployed in the CONNECTors. A Message Description Language (MDL) is the language used to describe a message format; the MDL specification for a particular protocol then describes its set of messages only. Message composers and parsers are implemented as general interpreters that execute the message description language specifications that are loaded. For example, a parser that interprets an SLP MDL instance will only parse SLP messages into the abstract message representation, i.e., it interprets the incoming message based upon the specification. Hence parsers are specialised to a particular protocol by associating the protocol specification to produce the Listener. Actuators are created using the same process to specialise generic message composers for text and binary protocols. An overview of this specialisation process is illustrated in Figure 8.

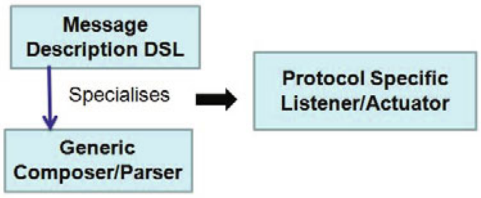


Fig. 8. The approach for generating Listeners and Actuators

There are a number of languages that can be used to parse network messages or parse data files. We investigated each of these as potential languages to be used in CONNECT; the results of this are seen in Table 1. It can be seen that a number of the tools focus solely on generating software to parse only data and messages, i.e., BinPac, Datascript and PacketTypes; therefore, these are unsuitable as it is equally important to be able to generate the composer part of the CONNECTor. Similarly, a number of the languages only consider binary data (i.e., all except PADS and ASN1.0); however, CONNECT requires the parsing of heterogeneous protocols which may use text or XML. In the example in Figure 6 SLP is a binary message, whereas SSDP is a text message. Hence, the only potential solutions are: i) PADS which offers the additional benefit of being able to infer data descriptions from received data [14], or ii)ASN 1.0. The drawback of these two are that they are not specifically designed for network packets, and we found when creating descriptions of example packet formats for SLP and GIOP that we were unable to successfully create the correct parsers and composers. Given

Table 1. Comparison of Data and Message Description Languages

Tool	Langauge	Generate Parser	Generate Composer	Domain
ASN 1.0 [31]	Java/C	x	x	Many encodings: binary, text, xml
BinPAC [27]	C++	x		Binary data and network packets
Datascript [2]	Java	x		Binary data
PADS [13]	C/ML	x	x	Binary or Text data
PacketTypes [23]	ML	x		Binary network packets
Melange [22]	ML	x	x	Binary network packets

the results of this investigation, Connect proposes new Message Description Languages along with their corresponding tools in order to first provide a simple mechanism to parse and compose network packets.

Connect is flexible to allow different types of language to be used to specify message formats; each language is termed an MDL. This flexibility better supports the parsing and composing of a wide range of protocols. For example, specialised languages for binary messages, text messages and XML messages can be utilised. To illustrate the approach we present a language for binary messages, and then a language for text messages. It is important to identify here that the role of these languages is to extract the information into a representation that is usable within Connect; the languages themselves do not seek to understand the content of the message, nor are they concerned with the application semantics of the message. Take for example an RPC request message invoking an operation `Foo`, these languages can extract the value 'Foo' for the label 'operation' but cannot determine its purpose.

Binary MDL. For conciseness we consider one protocol, the Internet Inter-ORB Protocol (IIOP). This example also serves to illustrate in general how communication with any protocol can be achieved from a high-level specification of the message format. Figure 9 shows the specified message format of the IIOP protocol, which is a General Inter-ORB Protocol (GIOP) message as identified by[4] transported over a TCP connection. In this specification there are three important constructs that are employed to describe the general outline of the messages for one protocol:

– <`Types`> list the types of each individual field type, e.g., the `VersionMajor` field type is an integer value. Types are separated from the message specification in order for field types to be reusable across multiple messages.
– <`Header`> includes the message format of the header for the binary protocol messages. If a header specification is present this is common to every message in the protocol (only one Header can be defined). In this GIOP message both messages: `GIOPRequest` and `GIOPReply` have the defined header `GIOP`.
– <`Message`> describes the packet format for the body of a particular message. Each protocol will typically contain multiple message bodies, for example the

[4] http://www.omg.org/spec/CORBAe/20080201/GIOP.idl

```
<Types>
<Protocol:String[GIOP]><VersionMajor:Integer[1]>
<VersionMinor:Integer [2]><Reserved:null>
<Frag:Boolean><Endian:Boolean[f-Endian]>
<MessageType:Integer [f-MsgType]><RequestID:Integer[f-UniqueID]>
<MessageLength:Integer[f-MsgLength]><Response:Boolean[true]>
<ObjectKeyLength:Integer><ObjectKey:Octets>
<ParameterArray:CORBAParameters>
<EndTypes>

<Header:GIOP>
<Protocol:32><VersionMajor:8><VersionMinor:8>
<Reserved:8><MessageType:8><MessageLength:32>
<End:Header>

<Message:GIOPRequest>
<Rule:MessageType=0>
<RequestID:32><Response:8><Reserved:24>
<TargetAddress:32><ObjectKeyLength:32>
<ObjectKey:ObjectKeyLength><align:32>
<OperationLength:32><Operation:OperationLength>
<align:32><ContextListLength:32>
<ServiceContext:ContextListLength><align:64>
<ParameterArray:eof>
<End:Message>

<Message:GIOPReply>
<Rule:MessageType=1>
<RequestID:32><ReplyStatus:32><ContextListLength:32>
<ServiceContext:ContextListLength><align:64>
<ParameterArray:eof>
<End:Message>
```

Fig. 9. Partial view of the GIOP message description

IIOP protocol here contains message bodies for two GIOP messages: a GIOP request message, and a GIOP reply message.

Hence, <`Header`> and <`Message`> specify the content of the message headers and bodies. The information specified within these then describes the fine-grained field content. To do this, both headers and bodies are composed of <`label:size`> entries for each field in the message. The *size* is the length of the field content in bits. There is one special label: <`rule:field=value`>; this is used to relate the correct message body with the header. For example, the GIOP `GIOPRequest` message applies when the value of the `MessageType` field in the header equals zero.

Other interesting features of the <`Types`> specifications are functions and constant values. Functions can be defined on types using the `[f-method()]` construct, e.g., `[f-MsgLength]` in Figure 9 is a built in function to return the length of the composed message. They are generally useful for calculating values that must be composed when creating a message (rather than parsing), i.e., the named f-method is executed by the marshaller to get the value that must be written. Similarly, constants are values that can be composed directly by the marshaller with the given value, e.g., <`Protocol:String[GIOP]`> states the the `Protocol` field is always the value 'GIOP'.

Text MDL. Text based protocols are different from binary protocols and therefore, a new MDL is required to generate the Listeners and Actuators. We again use one example specification to highlight the features of the Text MDL; a subset of the messages within SSDP is specified in Figure 10. Like the binary approach there is a list of field labels with their corresponding types in the <`Types`> section and again <`Header`> and <`Body`> are used to describe the individual messages. The key difference in this language is that we utilise field deliminators rather than bit lengths to distinguish the length of the fields. For example in the <`Header`>, `<Method:32>` means that the field is terminated by the '32' ASCII character, i.e., a space. In the case where multiple characters are used to delimit we employ commas to list the character values e.g. `<Version:13,10>` is a backslash r followed by a backslash n.

Another important feature of text protocols is that they are typically self-describing, i.e., the field label as well as the value will form the content of the message. For example, a HTTP message may contain "Host:www.lancs.ac.uk"; this defines a field with a label Host and a value www.lancs.ac.uk. Hence, text protocols are not rigidly defined in terms of the fields and their order. To support this property we employ the <**Fields:** > construct; this will parse/compose a list of free form self-describing fields into their label, size and values. For example, <`Fields:13,10:58`> splits fields using the 13,10 delimitor, then it uses the 58 value (a colon) to split the field into its label and value. The label must relate to a type specified in the <`Types`> section.

```
<Types>
<Method:String>
    <URI:String>
    <HTTP Version:String>
    <MX:Integer>
    <MAN:String>
     ...
<EndTypes>

<Header:SSDP>
    <Method:32>
    <URI:32>
    <Version:13 ,10>
    <Fields:13 ,10 :58>
<End:Header>

<Message:SSDP_Search>
    <Rule:Method=M-SEARCH>
<End:Message>

<Message:SSDP_Response>
    <Rule:Method=HTTP/1.1>
<End:Message>

<Message:SSDP_Notify>
    <Rule:Method=NOTIFY>
<End:Message>
```

Fig. 10. Partial SSDP Message Description

4 CONNECT in Action

To demonstrate the potential of the Connect architecture we consider a single case within a distributed marketplace scenario. Consider a stadium where vendors are selling products such as popcorn, hot dogs, beer and memorabilia, and consumers can search for products and place an order with a vendor. Both merchants and consumers use mobile devices with wireless networks deployed in the stadium. Merchants publish product info which the consumers can browse through. When a consumer requests a product, the merchant gets a notification of the amount ordered and the location of the consumer, to which he can respond with a yes/no. Given the scale of the scenario there are many potential interoperability issues (e.g. due to the unpredictable application and middleware technologies employed by both vendors consumers), hence we look at just one particular vendor and consumer case:

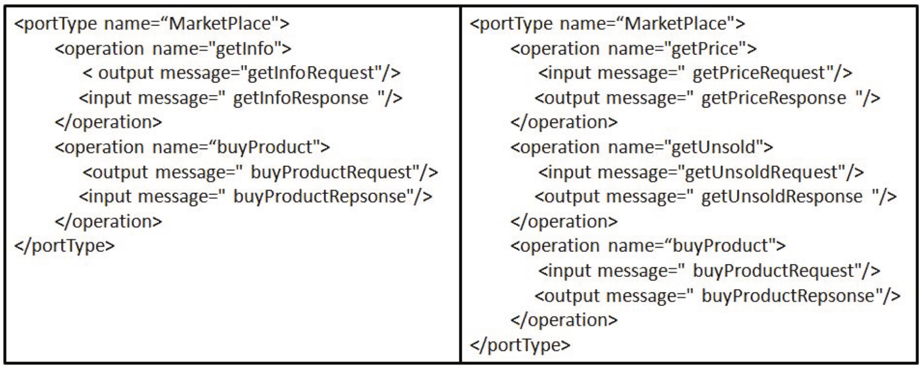
The client consumer application uses UPnP to perform lookup requests for nearby vendors, and then a message-based communication protocol (in this case SOAP) to interact with the found vendor, while the service merchant advertises their services using SLP and then employs an RPC-based protocol for communication with client (in this case CORBA, more specifically the IIOP protocol).

We apply the Connect architecture to build a Connector that allows the consumer to interact with the vendor in the face of this heterogeneity.

4.1 Phase 1: Discovery

The discovery enabler first monitors the running systems, and receives the UPnP lookup requests that describe the consumer application's requirements. It also receives the notification messages from the vendor in SLP that advertise the provided interface. The two plug-ins for the discovery enabler (SLP and UPnP plug-ins) listen on the appropriate multicast addresses: 239.255.255.253 port 427 for SLP, and 239.255.255.250 port 1900 for UPnP. These plug-ins then transform the content of the messages into both the affordance and interface descriptions (WSDL specifications) of the two networked systems as per the requirements of the Networked System Model. From this, the initial matchmaking is performed and given the similarity of application and operations provided– the two systems are determined to match, and it is now the objective to build a Connector that will allow the two to interact. A partial view of the two WSDL descriptions is shown in Figure 11. It is important to observe here that the two share the same data schema and thus we don't investigate here how Connect resolves data heterogeneity problem.

The discovery process also determines how these abstract operations are bound to concrete middleware protocols. In the consumer case they are bound to the SOAP asynchronous message protocol; here each of the messages that form an operation are sent asynchronously, point to point between the peers, e.g., the `buyProductRequest` of the `buyProduct` operation is sent as a SOAP message to the vendor. In the vendor case, the abstract operations are bound to the IIOP synchronous RPC protocol; here, the two messages that form the



```
<portType name="MarketPlace">
      <operation name="getInfo">
          < output message="getInfoRequest"/>
         <input message=" getInfoResponse "/>
      </operation>
      <operation name="buyProduct">
          <output message=" buyProductRequest"/>
         <input message=" buyProductRepsonse"/>
      </operation>
</portType>
```

```
<portType name="MarketPlace">
      <operation name="getPrice">
          <input message=" getPriceRequest"/>
         <output message=" getPriceResponse "/>
      </operation>
      <operation name="getUnsold">
          <input message="getUnsoldRequest"/>
         <output message=" getUnsoldResponse "/>
      </operation>
      <operation name="buyProduct">
          <input message=" buyProductRequest"/>
         <output message=" buyProductRepsonse"/>
      </operation>
</portType>
```

Fig. 11. WSDL interfaces for consumer (left) and vendor (right) marketplace applications

input and output of the operation are synchronously sent on the same transport connection, e.g., the `getPriceRequest` is received by the vendor who responds synchronously with the `getPriceResponse`.

4.2 Phase 2: Learning

The WSDL of the client and vendor in Figure 11 illustrate the heterogeneity of the two interfaces; they offer the same functionality, but do so with different behaviour. The next step in the CONNECT architecture is to learn the behaviours of these two systems. The learning enabler receives the WSDL documents from the discovery enabler and then interacts with a deployed instance of the CORBA vendor application in order to create the behaviour models for both the consumer and the vendor in this case. These are produced as LTS models and are illustrated for the SOAP consumer in Figure 12 and for the IIOP vendor in Figure 13. Here we can see that a vendor and consumer behaviour differs due to the heterogeneity of operations available from the interfaces. At this point we now have a completed Networked System Model (a description of the interface and behaviour of the

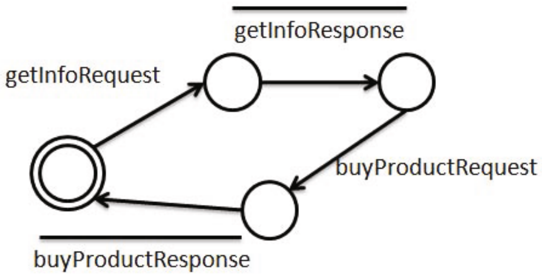
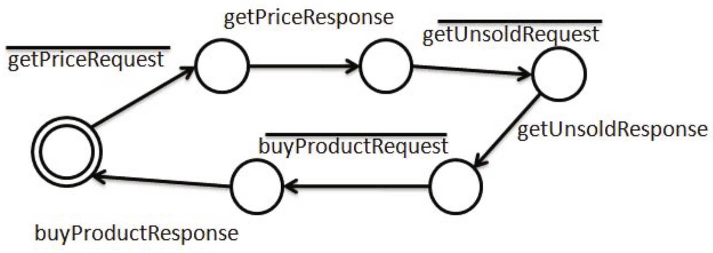


Fig. 12. LTS describing the behaviour of the SOAP consumer application



$\overline{\text{getPriceRequest}}$
getPriceResponse
$\overline{\text{getUnsoldRequest}}$
getUnsoldResponse
$\overline{\text{buyProductRequest}}$
buyProductResponse

Fig. 13. LTS describing the behaviour of the IIOP vendor application

system) for each of the two networked systems and can proceed to enable their interoperation.

4.3 Phase 3: Synthesis of a Mediator

The final step in the Connect process is to create the Connector that will mediate between the consumer's request and the merchant's response. To complete this the two LTS models are passed to the synthesis enabler. This performs two tasks:

– *Behaviour matching.* An ontology is provided for the domain that states where sequences of operations are equivalent, e.g., that the `getInfo` operation of the consumer and the `getPrice` combined with `getUnsold` of the vendor are equivalent. Further information about how the ontology-based behavioural matching is given in [4] [28].
– *Model synthesis.* The enabler produces an LTS that will mediate between the two systems; this LTS is shown in Figure 14. Here you can see how the interoperation is co-ordinated; the application differences and middleware differences are resolved as the mediator executes through each of the states of the LTS. Note, the transitions correspond to a message sent via a particular middleware protocol, i.e., either SOAP or IIOP as indicated by the dashed line here.

A Connector is then realised by using a model to code transformation to generate an executable mediator that can be deployed in the network between the two networked systems. The mediator for the SOAP and IIOP applications is seen in Figure 14. Here, the protocol messages are sent or received as per the protocol specification (the dotted line indicates that these are IIOP message, while the complete line indicates it is the SOAP protocol). Hence, the use of appropriate listeners and actuators (specific to the protocol) as described in Section 2.4 overcomes the problem of middleware heterogeneity, whereas the mediated sequence of application messages overcomes the application heterogeneity.

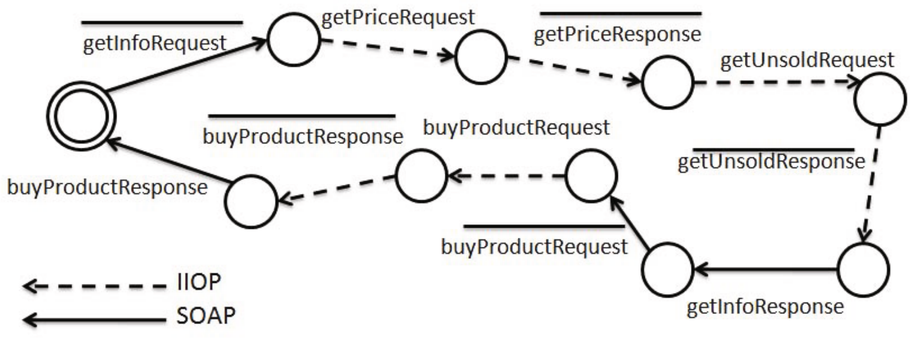

$\overline{\text{getInfoRequest}}$
getPriceRequest
$\overline{\text{getPriceResponse}}$
getUnsoldRequest
$\overline{\text{getUnsoldResponse}}$
getInfoResponse
$\overline{\text{buyProductRequest}}$
buyProductRequest
$\overline{\text{buyProductResponse}}$
buyProductResponse
IIOP
SOAP

Fig. 14. LTS describing the mediated interaction between the two systems

5 Conclusions and Future Perspectives

5.1 Concluding Remarks

The overall aim of CONNECT is to bridge the interoperability gap that results from the use of different data and protocols by the different entities involved in the software stack such as applications, middleware, platforms, etc. This aim is particularly targeted at heterogeneous, dynamic environments where systems must interact spontaneously, i.e., they only discover each other at runtime. This chapter has presented the CONNECT architecture to meet this particular objective; here we have seen how software enablers co-ordinate to create CONNECTors that translate between the heterogeneous legacy protocols.

This chapter has examined the problem of communicating with legacy protocols in further detail, and has shown how domain specific languages that describe message formats (MDLs) can be used to generate the required middleware dynamically. These listeners and actuators can receive and send messages that correspond to the protocol specification and therefore are able to address the heterogeneity of middleware protocols. Subsequently the generated software mediators that co-ordinate the operation of listeners and actuators are able to handle the variations in application operations (as shown in Section 4). For deeper insight into how these mediators are specified and created, the interested reader is pointed to following:

– [19] offers a comprehensive description of how CONNECT leverages active learning to determine the behaviour of a networked system.
– [4] examines the generation of CONNECTors in greater detail, illustrating the matching and mapping of networked system models and also describing the code generation techniques utilised.

We now discuss interesting directions for future research. Some of these are being actively pursued within the CONNECT project, whereas some are more general areas of research that can add to the understanding of interoperability solutions.

5.2 Future Research Direction: Advanced Learning of Middleware Protocols

In terms of *advanced learning*, we envisage further investigation of the role learning occupies within the architecture. At present, learning is focused solely on the behaviour model from the networked system model; that is, it aims to identify the application behaviour of a system. While this is important to the automation of Connectors, it only focuses on part of the behaviour. At present, the middleware protocol behaviour and their corresponding message formats must be defined (and be known by Connect) in advance. If a new system employs a novel protocol then Connect is unable to resolve the interoperability, hence the approach is not future proof. Rather it is required that we equally apply learning approaches at the middleware level; this would not be executed as frequently (e.g. within the flow of the Connect process) because a new protocol need only be learned once. There has been interesting work in the learning of message formats and communication protocol sequences for the purpose of network security, examples include: Polyglot [9], Tupni [11], and Autoformat [21] which employ binary analysis techniques to extract information about the protocols by observing the binary executables in action; these have the potential to form the basis of the learning the MDL specifications automatically. However, they remain limited to understanding the content of a message, what it does and what the purpose of the individual fields are–they can only deduce the field boundaries; hence, further research could look at the automated understanding of protocol content (which is potentially very important to understand if two protocols are compatible for interoperation).

5.3 Future Research Direction: The Role of Ontologies in Interoperability Frameworks

While only briefly discussed here, ontologies have an important role in the Connect architecture. Ontologies have been successfully employed within Web 2.0 applications, however these have only really considered the top level concerns such as discovering semantically similar systems. Connect is pushing the role of ontologies further, and is investigating going deep with the use of ontologies, i.e., using them at both the middleware and application level. Hence, to achieve better interoperability solutions ontologies cross-cut all of the Connect functions and enablers. This work is in the initial stages: this chapter has introduced the role of ontologies in the discovery, matching, and synthesis of Connectors rather than explain the methods; here, ontologies feature in the networked model and are employed in discovery and matching of affordances and descriptions, while matching of systems (including alignment based upon ontologies) leads to the synthesis of Connectors. In this domain an exciting area of future work is the application of ontologies to the lowest level of the Connect architecture, i.e. the interoperation between middleware protocols; ontologies can be applied to classify (discover the behaviour of) new network protocols and then use this to determine the low-level interoperability bridges (i.e., the matching and mapping of data field content between protocol messages, for example the matching of the 'methodName' field in XML-RPC with the operation field in IIOP and the subsequent translation of the data between the two–one is a null terminated string, the other isn't).

5.4 Future Research Direction: Interoperability Considering Non-functional Requirements

Networked systems also have *non-functional properties* and requirements which must be considered in order to ensure correct interoperation between networked systems. Future work should place equal importance on these requirements. To underpin this, this will first involve extracting the non-functional requirements from networked systems and adding these to the interface description in the Networked System Model. This will involve extending the discovery process to discover non-functional descriptions of the systems which are also published using discovery protocols. Finally, the Connectors must maintain particular non-functional requirements, e.g., dependability, security and trust are important and diverse properties within networked systems that must be maintained by an interoperability solution (and are particularly important in pervasive environments). Future research in this direction must consider solutions to correctly ensure that the interoperability solutions meets any of these domain requirements.

References

1. Adams, D.: The Hitchhiker's Guide To The Galaxy. Pan Books (1979)
2. Back, G.: Datascript - a specification and scripting language for binary data. In: Batory, D., Blum, A., Taha, W. (eds.) GPCE 2002. LNCS, vol. 2487, pp. 66–77. Springer, Heidelberg (2002)
3. Bertolino, A., Calabro, A., Di Giandomenico, F., Nostro, N.: Dependability and Performance Assessment of Dynamic CONNECTed Systems. In: Bernardo, M., Issarny, V. (eds.) SFM 2011. LNCS, vol. 6659, pp. 350–392. Springer, Heidelberg (2011)
4. Bertolino, A., Inverardi, P., Issarny, V., Sabetta, A., Spalazzese, R.: On-the-fly interoperability through automated mediator synthesis and monitoring. In: Margaria, T., Steffen, B. (eds.) ISoLA 2010, Part II. LNCS, vol. 6416, pp. 251–262. Springer, Heidelberg (2010)
5. Blair, G., Paolucci, M., Grace, P., Georgantas, N.: Interoperability in Complex Distributed Systems. In: Bernardo, M., Issarny, V. (eds.) SFM 2011. LNCS, vol. 6659, pp. 350–392. Springer, Heidelberg (2011)
6. Booth, D., Haas, H., McCabe, F., Newcomer, E., Champion, M., Ferris, C., Orchard, D.: Web services architecture. In: W3C (February 2004), `http://www.w3.org/TR/sawsdl/`
7. Brodie, M.: The long and winding road to industrial strength semantic web services. In: Proceedings of the 2nd International Semantic Web Conference (ISWC 2003) (October 2003)

8. Bromberg, Y.-D., Issarny, V.: INDISS: Interoperable discovery system for networked services. In: Alonso, G. (ed.) Middleware 2005. LNCS, vol. 3790, pp. 164–183. Springer, Heidelberg (2005)
9. Caballero, J., Yin, H., Liang, Z., Song, D.: Polyglot: automatic extraction of protocol message format using dynamic binary analysis. In: Proceedings of the 14th ACM Conference on Computer and Communications Security, CCS 2007, pp. 317–329. ACM, New York (2007)
10. Christensen, E., Curbera, F., Meredith, G., Weerawarana, S.: Web services description language (wsdl) 1.1 (March 2001), `http://www.w3.org/TR/wsdl`
11. Cui, W., Peinado, M., Chen, K., Wang, H.J., Irun-Briz, L.: Tupni: automatic reverse engineering of input formats. In: Proceedings of the 15th ACM Conference on Computer and Communications Security, CCS 2008, pp. 391–402. ACM, New York (2008)
12. Farrell, J., Lausen, H.: Semantic annotations for wsdl and xml schema (August 2007), `http://www.w3.org/TR/sawsdl/`
13. Fisher, K., Mandelbaum, Y., Walker, D.: The next 700 data description languages. In: Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, pp. 2–15. ACM, New York (2006)
14. Fisher, K., Walker, D., Zhu, K.Q., White, P.: From dirt to shovels: fully automatic tool generation from ad hoc data. In: Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, pp. 421–434. ACM, New York (2008)
15. UPnP Forum. Upnp device architecture version 1.0. (October 2008), `http://www.upnp.org/specs/arch/UPnP-arch-DeviceArchitecture-v1.0.pdf`
16. Grace, P., Blair, G., Samuel, S.: A reflective framework for discovery and interaction in heterogeneous mobile environments. ACM SIGMOBILE Mobile Computing and Communications Review 9(1), 2–14 (2005)
17. Gudgin, M., Hadley, M., Mendelsohn, N., Moreau, J., Frystyk Nielsen, H., Karmarkar, A., Lafon, Y.: Soap version 1.2 part 1: Messaging framework (April 2001), `http://www.w3.org/TR/soap12-part1`
18. Guttman, E., Perkins, C., Veizades, J.: Service location protocol version 2, IETF RFC 2608 (June 1999), `http://www.ietf.org/rfc/rfc2608.txt`
19. Howar, F., Jonsson, B., Merten, M., Steffen, B., Cassel, S.: On handling data in automata learning - considerations from the connect perspective. In: Margaria, T., Steffen, B. (eds.) ISoLA 2010. LNCS, vol. 6416, pp. 221–235. Springer, Heidelberg (2010)
20. Issarny, V., Bennaceur, A., Bromberg, Y.-D.: Middleware-layer CONNECTor Synthesis. In: Bernardo, M., Issarny, V. (eds.) SFM 2011. LNCS, vol. 6659, pp. 350–392. Springer, Heidelberg (2011)
21. Lin, Z., Jiang, X., Xu, D., Zhang, X.: Automatic protocol format reverse engineering through conectect-aware monitored execution. In: 15th Symposium on Network and Distributed System Security (NDSS) (2008)
22. Madhavapeddy, A., Ho, A., Deegan, T., Scott, D., Sohan, R.: Melange: creating a "functional" internet. In: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007, EuroSys 2007, pp. 101–114. ACM, New York (2007)
23. McCann, P.J., Chandra, S.: Packet types: abstract specification of network protocol messages. In: Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, SIGCOMM 2000, pp. 321–333. ACM, New York (2000)

24. Nakazawa, J., Tokuda, H., Edwards, W.K., Ramachandran, U.: A bridging framework for universal interoperability in pervasive systems. In: Proceedings of the 26th IEEE International Conference on Distributed Computing Systems, ICDCS 2006. IEEE Computer Society, Washington, DC, USA (2006)
25. OASIS. Web services dynamic discovery (wsdiscovery) version 1.1. (July 2009), `http://docs.oasis-open.org/ws-dd/discovery/1.1/os/wsdd-discovery-1.1-spec-os.pdf`
26. OMG. The common object request broker: Architecture and specification version 2.0. Technical report, Object Management Group (1995)
27. Pang, R., Paxson, V., Sommer, R., Peterson, L.: binpac: a yacc for writing application protocol parsers. In: Proceedings of the 6th ACM SIGCOMM Conference on Internet Measurement, IMC 2006, pp. 289–300. ACM, New York (2006)
28. Spalazzese, R., Inverardi, P., Tivoli, M.: Application-layer CONNECTor Synthesis. In: Bernardo, M., Issarny, V. (eds.) SFM 2011. LNCS, vol. 6659, pp. 350–392. Springer, Heidelberg (2011)
29. Raffelt, H., Steffen, B.: LearnLib: A library for automata learning and experimentation. In: Baresi, L., Heckel, R. (eds.) FASE 2006. LNCS, vol. 3922, pp. 377–380. Springer, Heidelberg (2006)
30. Roman, M., Kon, F., Campbell, R.H.: Reflective middleware: From your desk to your hand. IEEE Distributed Systems Online 2 (May 2001)
31. Tantiprasut, D., Neil, J., Farrell, C.: Asn.1 protocol specification for use with arbitrary encoding schemes. IEEE/ACM Trans. Netw. 5, 502–513 (1997)

Automated Verification Techniques for Probabilistic Systems

Vojtěch Forejt[1], Marta Kwiatkowska[1], Gethin Norman[2], and David Parker[1]

[1] Oxford University Computing Laboratory, Parks Road, Oxford, OX1 3QD, UK
[2] School of Computing Science, University of Glasgow, Glasgow, G12 8RZ, UK

Abstract. This tutorial provides an introduction to probabilistic model checking, a technique for automatically verifying quantitative properties of probabilistic systems. We focus on Markov decision processes (MDPs), which model both stochastic and nondeterministic behaviour. We describe methods to analyse a wide range of their properties, including specifications in the temporal logics PCTL and LTL, probabilistic safety properties and cost- or reward-based measures. We also discuss multi-objective probabilistic model checking, used to analyse trade-offs between several different quantitative properties. Applications of the techniques in this tutorial include performance and dependability analysis of networked systems, communication protocols and randomised distributed algorithms. Since such systems often comprise several components operating in parallel, we also cover techniques for compositional modelling and verification of multi-component probabilistic systems. Finally, we describe three large case studies which illustrate practical applications of the various methods discussed in the tutorial.

1 Introduction

Many computerised systems exhibit probabilistic behaviour. Messages transmitted across wireless networks, for example, may be susceptible to losses and delays, or system components may be prone to failure. In both cases, probability is a valuable tool for the modelling and analysis of such systems. Another source of stochastic behaviour is the use of randomisation, for example to break symmetry or prevent flooding in communication networks. This is an integral part of wireless communication protocols such as Bluetooth or Zigbee. Randomisation is also a useful tool in security protocols, for example to guarantee anonymity, and in the construction of dynamic power management schemes.

Formal verification is a systematic approach that applies mathematical reasoning to obtain guarantees about the correctness of a system. One successful method in this domain is *model checking*. This is based on the construction and analysis of a system model, usually in the form of a finite state automaton, in which states represent the possible configurations of the system and transitions between states capture the ways that the system can evolve over time. Desired properties such as "no two threads obtain a lock simultaneously" or "the system always eventually delivers an acknowledgement to a request" are then expressed in temporal logic and the model is analysed in an automatic fashion to determine whether or not the model satisfies the properties.

There is increasing interest in the development of *quantitative* verification techniques, which take into account probabilistic and timed aspects of a system. *Probabilistic model checking*, for example, is a generalisation of model checking that builds and analyses probabilistic models such as Markov chains and Markov decision processes. A wide range of quantitative properties of these systems can be expressed in probabilistic extensions of temporal logic and systematically analysed against the constructed model. These properties can capture many different aspects of system behaviour, from reliability, e.g. "the probability of an airbag failing to deploy on demand", to performance, e.g. "the expected time for a network protocol to successfully send a message packet".

This tutorial gives an introduction to probabilistic model checking for *Markov decision processes*, a commonly used formalism for modelling systems that exhibit a combination of probabilistic and nondeterministic behaviour. It covers the underlying theory, discusses probabilistic model checking algorithms and their implementation, and gives an illustration of the application of these techniques to some large case studies. The tutorial is intended to complement [67], which focuses on probabilistic model checking for discrete- and continuous-time Markov chains, rather than Markov decision processes. There is also an accompanying website [91], providing models for the PRISM probabilistic model checker [56] that correspond to the various running examples used throughout and to the case studies in Section 10.

There are many other good references relating to the material covered in this tutorial and we provide pointers throughout. In particular, Chapter 10 of [11] covers some of the MDP model checking techniques presented here, but in greater depth, with additional focus on the underlying theory and proofs. We also recommend the theses by Segala [80], de Alfaro [1] and Baier [7], which provide a wealth of in-depth material on a variety of topics relating to MDPs, along with detailed pointers to other relevant literature. Finally, although not focusing on verification, [76] is an excellent general reference on MDPs.

Outline. The tutorial is structured as follows. We begin, in Section 2, with background material on probability theory and discrete-time Markov chains. In Section 3, we introduce the model of Markov decision processes. Section 4 describes the computation of a key property of MDPs, probabilistic reachability, and Section 5 covers reward-based properties. Section 6 concerns how to formally specify properties of MDPs using the probabilistic temporal logic PCTL, and shows how the techniques introduced in the previous sections can be used to perform model checking. Section 7 describes how to verify MDPs against safety properties and the logic LTL using automata-based techniques. Two advanced topics, namely multi-objective probabilistic model checking and compositional probabilistic verification, are the focus of Sections 8 and 9. In Section 10, we list some of the software tools available for model checking MDPs and describe three illustrative case studies. Section 11 concludes by discussing active research areas and suggesting further reading.

2 Background Material

2.1 Probability Distributions and Measures

We begin by briefly summarising some definitions and notations relating to probability distributions and measures. We assume that the reader has some familiarity with basic probability theory. Good introductory texts include [19,42].

Definition 1 (Probability distribution). *A (discrete)* probability distribution *over a countable set S is a function $\mu : S \rightarrow [0,1]$ satisfying $\sum_{s \in S} \mu(s){=}1$.*

We use $[s_0{\mapsto}x_0, \ldots, s_n{\mapsto}x_n]$ to denote the distribution that chooses s_i with probability x_i for all $0{\leqslant}i{\leqslant}n$ and $\mathit{Dist}(S)$ for the set of distributions over S. The *point distribution* on $s \in S$, denoted $[s{\mapsto}1]$, is the distribution that assigns probability 1 to s. Given two distributions $\mu_1 \in \mathit{Dist}(S_1)$ and $\mu_2 \in \mathit{Dist}(S_2)$, the *product distribution* $\mu_1{\times}\mu_2 \in \mathit{Dist}(S_1{\times}S_2)$ is defined by $\mu_1{\times}\mu_2((s_1, s_2)) = \mu_1(s_1){\cdot}\mu_2(s_2)$.

Definition 2 (Probability space). *A* probability space *over a sample space Ω is a triple $(\Omega, \mathcal{F}, Pr)$, where $\mathcal{F} \subseteq 2^{\Omega}$ is a* σ-algebra *over Ω, i.e.*

- *$\varnothing, \Omega \in \mathcal{F}$;*
- *if $A \in \mathcal{F}$, then $\Omega \setminus A \in \mathcal{F}$;*
- *if $A_i \in \mathcal{F}$ for all $i \in \mathbb{N}$, then $\cup_{i \in \mathbb{N}} A_i \in \mathcal{F}$*

and $Pr : \mathcal{F} \rightarrow [0,1]$ is a probability measure *over $(\Omega, \mathcal{F})$, i.e.*

- *$Pr(\varnothing) = 0$ and $Pr(\Omega) = 1$;*
- *$Pr(\cup_{i \in \mathbb{N}} A_i) = \sum_{i \in \mathbb{N}} Pr(A_i)$ for all countable pairwise disjoint sequences $A_1, A_2, \ldots$ of $\mathcal{F}$.*

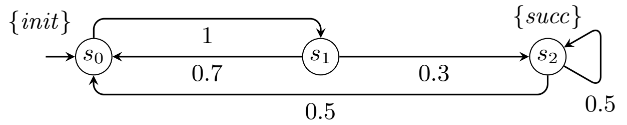
Sets contained in the σ-algebra $\mathcal{F}$ are said to be *measurable.* A (non-negative) *random variable* over a probability space $(\Omega, \mathcal{F}, Pr)$ is a *measurable function* $X : \Omega \rightarrow \mathbb{R}_{\geqslant 0}$, i.e. a function such that $X^{-1}([0, r]) \in \mathcal{F}$ for all $r \in \mathbb{R}_{\geqslant 0}$. The *expected value* of X with respect to Pr is given by the following integral:

$$\mathbb{E}[X] \overset{\text{def}}{=} \int_{\omega \in \Omega} X(\omega) \, \mathrm{d}Pr \; .$$

2.2 Discrete-Time Markov Chains

Next, we introduce the model of *discrete-time Markov chains* (DTMCs). We provide just a brief overview of DTMCs, as required for the remainder of this tutorial. For more in-depth coverage of the topic, we recommend the textbooks by Stewart [83] and Kulkarni [65]. For a tutorial on probabilistic model checking for Markov chains, see for example [67].

Definition 3 (Discrete-time Markov chain). *A* discrete-time Markov chain *(DTMC) is a tuple $\mathcal{D} = (S, \overline{s}, \mathbf{P}, L)$ where S is a (countable) set of states, $\overline{s} \in S$ is an initial state, $\mathbf{P} : S{\times}S \rightarrow [0,1]$ is a transition probability matrix such that $\sum_{s' \in S} \mathbf{P}(s, s') = 1$ for all $s \in S$, and $L : S \rightarrow 2^{AP}$ is a labelling function mapping each state to a set of atomic propositions taken from a set AP.*



$$\mathbf{P} = \begin{pmatrix} 0 & 1 & 0 \\ 0.7 & 0 & 0.3 \\ 0.5 & 0 & 0.5 \end{pmatrix}$$

Fig. 1. An example DTMC and its transition probability matrix **P**

One way to view a DTMC $\mathcal{D}{=}(S, \overline{s}, \mathbf{P}, L)$ is as a state-transition system in which transitions are augmented with probabilities indicating their likelihood. From each state $s \in S$, the probability of a transition to s' occurring is $\mathbf{P}(s, s')$.

A *path* represents one possible execution of $\mathcal{D}$. Formally, a (finite or infinite) path of $\mathcal{D}$ is a sequence of states $s_0 s_1 s_2 \ldots$ such that $\mathbf{P}(s_i, s_{i+1}){>}0$ for all $i{\geqslant}0$. We use $\mathit{FPath}_{\mathcal{D},s}$ and $\mathit{IPath}_{\mathcal{D},s}$, respectively, to denote the set of all finite and infinite paths starting from state s of $\mathcal{D}$.

In order to reason formally about the behaviour of $\mathcal{D}$, we need to determine the probability that certain paths are taken. We proceed by constructing, for each state $s \in S$, a *probability space* over the set of infinite paths $\mathit{IPath}_{\mathcal{D},s}$. This is outlined below and for further details see [64]. The basis of the construction is the probability of individual finite paths induced by the transition probability matrix $\mathbf{P}$. More precisely, the probability of the path $\rho{=}s_0 \ldots s_n$ is given by $\mathbf{P}(\rho) \overset{\text{def}}{=} \prod_{i=0}^{n-1} \mathbf{P}(s_i, s_{i+1})$. We begin by defining, for each finite path $\rho \in \mathit{FPath}_{\mathcal{D},s}$, the *basic cylinder* C_ρ that consists of all infinite paths starting with ρ. Using properties of cylinders, we can then construct the probability space $(\mathit{IPath}_{\mathcal{D},s}, \mathcal{F}_{\mathcal{D},s}, \mathit{Pr}_{\mathcal{D},s})$ where $\mathcal{F}_{\mathcal{D},s}$ is the smallest σ-algebra generated by the basic cylinders $\{C_\rho \,|\, \rho \in \mathit{FPath}_{\mathcal{D},s}\}$ and $\mathit{Pr}_{\mathcal{D},s}$ is the unique measure such that $\mathit{Pr}_{\mathcal{D},s}(C_\rho) = \mathbf{P}(\rho)$ for all $\rho \in \mathit{FPath}_{\mathcal{D},s}$.

Example 1. Consider the 3-state DTMC $\mathcal{D}{=}(S, \overline{s}, \mathbf{P}, L)$ of Figure 1. Here, $S{=}\{s_0, s_1, s_2\}$, $\overline{s}{=}s_0$, the transition probability matrix $\mathbf{P}$ is shown in Figure 1, $L(s_0){=}\{\mathit{init}\}$, $L(s_1){=}\varnothing$ and $L(s_2){=}\{\mathit{succ}\}$. We have, for example:

– $\mathit{Pr}_{\mathcal{D},s_0}(\{\pi \text{ starts } s_0 s_1 s_2 s_0\}) = 1{\cdot}0.3{\cdot}0.5 = 0.15$;
– $\mathit{Pr}_{\mathcal{D},s_0}(\{(s_0 s_1 s_2)^\omega\}) = \lim_{n\to\infty} \mathit{Pr}_{\mathcal{D},s_0}(\{\pi \text{ starts } (s_0 s_1 s_2)^n\})$
$= \lim_{n\to\infty} 1{\cdot}0.3{\cdot}(0.5{\cdot}1{\cdot}0.3)^{n-1} = 0$;
– $\mathit{Pr}_{\mathcal{D},s_0}(\{\pi \text{ contains } s_2\}) = \sum_{n=1}^{\infty} \mathit{Pr}_{\mathcal{D},s_0}(\{\pi \text{ starts } (s_0 s_1)^n s_2\})$
$= \sum_{n=1}^{\infty} 1{\cdot}(0.7{\cdot}1)^{n-1}{\cdot}0.3 = 1.$ ■

3 Markov Decision Processes

This tutorial focuses on the model of *Markov decision processes* (MDPs), which are a widely used formalism for modelling systems that exhibit both *probabilistic* and *nondeterministic* behaviour. From the point of view of applying *quantitative verification*, nondeterminism is an essential tool to capture several different aspects of system behaviour:

– *unknown environment*: if the system interacts with other components whose behaviour is unknown, this can be modelled with nondeterminism;

– *concurrency*: in a distributed system comprising multiple components operating in parallel, nondeterminism is used to represent the different possible interleavings of the executions of the components;
– *underspecification*: if certain parts of a system are either unknown or too complex to be modelled efficiently, these can be abstracted away using nondeterminism.

Alternatively, we can use nondeterminism to capture the possible ways that a *controller* can influence the behaviour of the system. The multi-objective techniques that we describe in Section 8 can be seen as a way of performing *controller synthesis*. In a similar vein, MDPs are also widely used in domains such as planning and robotics. Formally, we define an MDP as follows.

Definition 4 (Markov decision process). *A* Markov decision process *(MDP) is a tuple $\mathcal{M}{=}(S, \overline{s}, \alpha_{\mathcal{M}}, \delta_{\mathcal{M}}, L)$ where S is a* finite *set of states, $\overline{s} \in S$ is an initial state, $\alpha_{\mathcal{M}}$ is a* finite *alphabet, $\delta_{\mathcal{M}} : S{\times}\alpha_{\mathcal{M}} \rightarrow Dist(S)$ is a (partial) probabilistic transition function and $L : S \rightarrow 2^{AP}$ is a labelling function mapping each state to a set of atomic propositions taken from a set AP.*

Transitions between states in an MDP $\mathcal{M}$ occur in two steps. First, a choice between one or more *actions* from the alphabet $\alpha_{\mathcal{M}}$ is made. The set of available actions in a state s is given by $A(s) \stackrel{\text{def}}{=} \{a \in \alpha_{\mathcal{M}} \mid \delta_{\mathcal{M}}(s, a) \text{ is defined}\}$. To prevent deadlocks, we assume that $A(s)$ is non-empty for all $s \in S$. The selection of an action $a \in A(s)$ is *nondeterministic*. Secondly, a successor state s' is chosen randomly, according to the probability distribution $\delta_{\mathcal{M}}(s, a)$, i.e. the probability that a transition to s' occurs equals $\delta_{\mathcal{M}}(s, a)(s')$.

An *infinite path* through an MDP is a sequence $\pi = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \cdots$ (occasionally written as $s_0 a_0 s_1 a_1 \ldots$) where $s_i \in S$, $a_i \in A(s_i)$ and $\delta_{\mathcal{M}}(s_i, a_i)(s_{i+1}){>}0$ for all $i \in \mathbb{N}$. A *finite path* $\rho = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \cdots \xrightarrow{a_{n-1}} s_n$ is a prefix of an infinite path ending in a state. We denote by $\mathit{FPath}_{\mathcal{M},s}$ and $\mathit{IPath}_{\mathcal{M},s}$, respectively, the set of all finite and infinite paths starting from state s of $\mathcal{M}$. We use $\mathit{FPath}_{\mathcal{M}}$ and $\mathit{IPath}_{\mathcal{M}}$ for the sets of *all* such paths in the MDP. Where the context is clear, we will drop the subscript $\mathcal{M}$. For a finite path $\rho = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \cdots \xrightarrow{a_{n-1}} s_n$, $|\rho| = n$ denotes its length and $\mathit{last}(\rho) = s_n$ its last state. For a (finite or infinite) path $\pi = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \cdots$, its ($i$+1)th state s_i is denoted $\pi(i)$ and its *trace*, $\mathit{tr}(\pi)$, is the sequence of actions $a_0 a_1 \ldots$ When, in later parts of this tutorial, we formalise ways to define properties of MDPs, we will use both *action-based* properties, based on path traces, and *state-based* properties, using the atomic propositions assigned to each state by the labelling function L.

A *reward structure* on an MDP is useful for representing quantitative information about the system the MDP represents, for example, the power consumption, number of packets sent, size of a queue or the number of lost requests. Formally, we define rewards on both the states and actions of an MDP as follows.

Definition 5 (Reward structure). *A* reward structure *for an MDP $\mathcal{M} = (S, \overline{s}, \alpha_{\mathcal{M}}, \delta_{\mathcal{M}}, L)$ is a tuple $r{=}(r_{\mathit{state}}, r_{\mathit{action}})$ comprising a* state reward function $r_{\mathit{state}} : S \rightarrow \mathbb{R}_{\geqslant 0}$ *and an* action reward function $r_{\mathit{action}} : S{\times}\alpha_{\mathcal{M}} \rightarrow \mathbb{R}_{\geqslant 0}$.

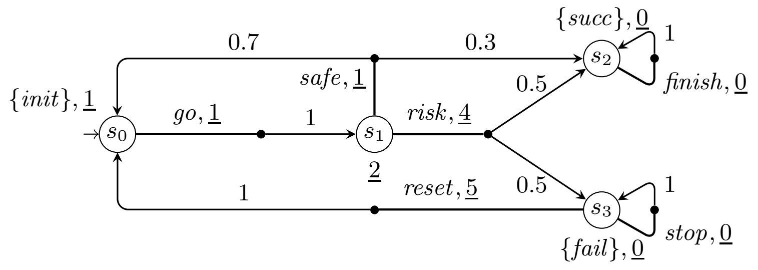


Fig. 2. A running example: an MDP, annotated with a reward structure

We consistently use the terminology *rewards* but, often, these will be used to model *costs*. The *action rewards* in a reward structure are also referred to elsewhere as *transition rewards*, *impulse rewards* or *state-action* rewards.

We next introduce the notion of *end components* which, informally, are parts of the MDP in which it possible to remain forever once entered.

Definition 6 (End component). *Let $\mathcal{M} = (S, \overline{s}, \alpha_{\mathcal{M}}, \delta_{\mathcal{M}}, L)$ be an MDP. An* end component *(EC) of $\mathcal{M}$ is a pair (S', δ') comprising a subset $S' \subseteq S$ of states and partial probabilistic transition function $\delta' : S' \times \alpha_{\mathcal{M}} \to Dist(S)$ satisfying the following conditions:*

- *(S', δ') defines a sub-MDP of $\mathcal{M}$, i.e. for all $s' \in S'$ and $a \in \alpha_M$, if $\delta'(s', a)$ is defined, then $\delta'(s', a) = \delta(s', a)$ and $\delta'(s', a)(s'') > 0$ only for states $s'' \in S'$;*
- *the underlying graph of (S', δ') is strongly connected.*

An EC (S', δ') is maximal *if there is no distinct EC (S'', δ'') such that for any $s \in S$ and $a \in \alpha_{\mathcal{M}}$, if $\delta'(s, a)$ is defined, then so is $\delta''(s, a)$.*

Algorithms to detect end components can be found in [1,11].

Example 2. Consider the MDP $\mathcal{M} = (S, \overline{s}, \alpha_{\mathcal{M}}, \delta_{\mathcal{M}}, L)$ from Figure 2. Here, $S = \{s_0, s_1, s_2, s_3\}$, $\overline{s} = s_0$ and $\alpha_{\mathcal{M}} = \{go, risk, safe, finish, stop, reset\}$. Considering the probabilistic transition function, for example, from state s_1 we have:

$$\delta_{\mathcal{M}}(s_1, risk) = [\, s_2 \mapsto 0.5,\ s_3 \mapsto 0.5 \,]$$
$$\delta_{\mathcal{M}}(s_1, safe) = [\, s_0 \mapsto 0.7,\ s_2 \mapsto 0.3 \,]$$

and $\delta(s_1, a)$ is undefined if $a \in \{go, finish, stop, reset\}$, i.e. $A(s_1) = \{risk, safe\}$. The labelling of e.g. states s_1 and s_2 is given by $L(s_1) = \varnothing$ and $L(s_2) = \{succ\}$.

The MDP models a system that aims to execute a task. After some routine initialisation (captured by the action *go*), there are two possibilities: it can either perform the task using a *safe* procedure, in which case the probability of finishing the task successfully is 0.3, but with probability 0.7 the system restarts; or, it can perform the task by a *risk*y procedure, in which case the probability of finishing the task is higher (0.5), but there is a 50% chance of complete failure, after which the system can only be restarted using the action *reset*.

State and action reward functions r_{state} and r_{action} are also depicted in the figure where the rewards are the underlined numbers next to states or action labels, e.g. $r_{state}(s_1)=2$ and $r_{action}(s_1, risk)=4$. An example of a finite path is $\rho = s_0 \xrightarrow{go} s_1 \xrightarrow{risk} s_3$ and an example of an infinite path is:

$$\pi = s_0 \xrightarrow{go} s_1 \xrightarrow{safe} s_0 \xrightarrow{go} s_1 \xrightarrow{safe} s_0 \xrightarrow{go} \cdots$$

The MDP contains two end components, namely $(\{s_2\}, \{(s_2, stop) \mapsto [s_2 \mapsto 1]\})$ and $(\{s_3\}, \{(s_3, stop) \mapsto [s_3 \mapsto 1]\})$. Both are maximal. ■

Adversaries. To reason formally about MDPs, in the way described for DTMCs in Section 2.2, we need a probability space over infinite paths. However, a probability space can only be constructed once all the nondeterminism present has been resolved. Each possible resolution of nondeterminism is represented by an *adversary*, which is responsible for choosing an action in each state of the MDP, based on the history of its execution so far. Adversaries are, depending on the context, often referred to by a variety of other names, including *strategies*, *schedulers* and *policies*.

Definition 7 (Adversary). *An* adversary *of an MDP $\mathcal{M} = (S, \overline{s}, \alpha_{\mathcal{M}}, \delta_{\mathcal{M}}, L)$ is a function $\sigma : FPath_{\mathcal{M}} \rightarrow Dist(\alpha_{\mathcal{M}})$ such that $\sigma(\rho)(a)>0$ only if $a \in A(last(\rho))$.*

In general, the choice of an action can be made randomly and depend on the full history of the MDP, but is limited to the actions available in the current state. The set of *all* adversaries of $\mathcal{M}$ is $Adv_{\mathcal{M}}$. There are several important classes of adversaries that we will now summarise. An adversary σ is *deterministic* if $\sigma(\rho)$ is a point distribution for all $\rho \in FPath_{\mathcal{M}}$, and *randomised* otherwise.

Definition 8 (Memoryless adversary). *An adversary σ is* memoryless *if $\sigma(\rho)$ depends only on $last(\rho)$, that is, for any $\rho, \rho' \in FPath_{\mathcal{M}}$ such that $last(\rho) = last(\rho')$, we have $\sigma(\rho) = \sigma(\rho')$.*

Definition 9 (Finite-memory adversary). *An adversary σ is* finite-memory *if there exists a tuple $(Q, \overline{q}, \sigma_u, \sigma_s)$ comprising:*

- *a finite set of* modes *Q;*
- *an initial mode $\overline{q} \in Q$;*
- *a mode update function $\sigma_u : Q \times \alpha_{\mathcal{M}} \times S \rightarrow Q$;*
- *an action selection function $\sigma_s : Q \times S \rightarrow Dist(\alpha_{\mathcal{M}})$*

such that $\sigma(\rho)=\sigma_s(\hat{\sigma}_u(\rho), last(\rho))$ for all $\rho \in FPath_{\mathcal{M}}$, where $\hat{\sigma}_u : FPath_{\mathcal{M}} \rightarrow Q$ is the function determined by $\hat{\sigma}_u(s)=\overline{q}$ and $\hat{\sigma}_u(\rho a s)=\sigma_u(\hat{\sigma}_u(\rho), a, s)$ for all $\rho \in FPath_{\mathcal{M}}$, $a \in \alpha_{\mathcal{M}}$, and $s \in S$.

Notice that a memoryless adversary is a finite-memory adversary with one mode.

Under a particular adversary σ, the behaviour of an MDP $\mathcal{M}$ is fully probabilistic and can be captured by a (countably infinite-state) discrete-time Markov chain, denoted $\mathcal{M}_\sigma$, each state of which is a finite path of $\mathcal{M}$.

Definition 10 (Induced DTMC). *For an MDP $\mathcal{M} = (S, \overline{s}, \alpha_{\mathcal{M}}, \delta_{\mathcal{M}}, L)$ and adversary $\sigma \in Adv_{\mathcal{M}}$, the* induced DTMC *is $\mathcal{M}_\sigma = (T, \overline{s}, \mathbf{P}, L')$ where:*

– $T = \mathit{FPath}_{\mathcal{M}}$;
– *for any* $\rho, \rho' \in \mathit{FPath}_{\mathcal{M}}$:

$$\mathbf{P}(\rho, \rho') = \begin{cases} \sigma(\rho)(a) \cdot \delta_{\mathcal{M}}(\mathit{last}(\rho), a)(s) & \text{if } \rho' = \rho a s,\ a \in A(\rho) \text{ and } s \in S \\ 0 & \text{otherwise;} \end{cases}$$

– $L'(\rho) = L(\mathit{last}(\rho))$ *for all* $\rho \in \mathit{FPath}_{\mathcal{M}}$.

The induced DTMC $\mathcal{M}_\sigma$ has an infinite number of states. However, in the case of finite-memory adversaries (and hence also the subclass of memoryless adversaries), we can also construct a finite-state *quotient DTMC*. More precisely, if the finite-memory adversary is defined by the tuple $(Q, \overline{q}, \sigma_u, \sigma_s)$, then we stipulate two paths ρ and ρ' to be equivalent whenever they get mapped to the same mode, i.e. $\hat{\sigma}_u(\rho){=}\hat{\sigma}_u(\rho')$, and the last action and state of ρ and ρ' are the same. It follows that we can classify equivalence classes of paths by tuples of the form (q, a, s), where q is the mode that paths of the equivalence class get mapped to and a and s are the last action and state of these paths. Formally, the finite-state quotient can be defined as follows (since the path consisting of just the initial state does not have a "last" action, we introduce a new symbol $\perp$).

Definition 11 (Quotient DTMC). *The* quotient DTMC *for an MDP* $\mathcal{M} = (S, \overline{s}, \alpha_{\mathcal{M}}, \delta_{\mathcal{M}}, L)$ *and finite-memory adversary* σ *defined by the tuple* $(Q, \overline{q}, \sigma_u, \sigma_s)$ *is the finite-state DTMC* $\mathcal{M}^q_\sigma = (T, \overline{s}', \mathbf{P}, L')$ *where:*

– $T = (Q \times \alpha_{\mathcal{M}} \times S) \cup \{(\overline{q}, \perp, \overline{s})\}$;
– $\overline{s}' = (\overline{q}, \perp, \overline{s})$;
– $\mathbf{P}((q, a, s), (q', a', s')) = \sigma_s(q, s)(a') \cdot \delta_{\mathcal{M}}(s, a')(s')$ *whenever* $q' = \sigma_u(q, a', s')$ *and equals* 0 *otherwise;*
– $L'((q, a, s)) = L(s)$.

Probability Spaces. For a given (general) adversary σ, we associate the infinite paths in $\mathcal{M}$ and $\mathcal{M}^\sigma$ by defining the following bijection f:

$$f(s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \cdots) \overset{\text{def}}{=} (s_0)\,(s_0 a_0 s_1)\,(s_0 a_0 s_1 a_1 s_2)\,\ldots$$

for all infinite paths $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \cdots \in \mathit{IPath}_{\mathcal{M}}$. Now, for any state s of $\mathcal{M}$, using this function and the probability measure $\mathit{Pr}_{\mathcal{M}_\sigma, s}$ given in Section 2.2 for the DTMC $\mathcal{M}_\sigma$, we can define a probability measure $\mathit{Pr}^\sigma_{\mathcal{M},s}$ over $\mathit{IPath}_{\mathcal{M},s}$ capturing the behaviour of $\mathcal{M}$ from state s under adversary σ. Furthermore, for a random variable $X : \mathit{IPath}_{\mathcal{M},s} \to \mathbb{R}_{\geqslant 0}$, we can compute the expected value of X from state s in $\mathcal{M}$ under adversary σ, which we denote by $\mathbb{E}^\sigma_{\mathcal{M},s}(X)$.

In practice, we are mainly interested in *minimising* or *maximising* either the probability of a certain set of paths, or the expected value of some random variable. Therefore, for any measurable set of paths $\Omega \subseteq \mathit{IPath}_{\mathcal{M},s}$ and random variable $X : \mathit{IPath}_{\mathcal{M},s} \to \mathbb{R}_{\geqslant 0}$, we define:

$$\begin{aligned} \mathit{Pr}^{\min}_{\mathcal{M},s}(\Omega) &\overset{\text{def}}{=} \inf\nolimits_{\sigma \in \mathit{Adv}_{\mathcal{M}}} \mathit{Pr}^\sigma_{\mathcal{M},s}(\Omega) \\ \mathit{Pr}^{\max}_{\mathcal{M},s}(\Omega) &\overset{\text{def}}{=} \sup\nolimits_{\sigma \in \mathit{Adv}_{\mathcal{M}}} \mathit{Pr}^\sigma_{\mathcal{M},s}(\Omega) \\ \mathbb{E}^{\min}_{\mathcal{M},s}(X) &\overset{\text{def}}{=} \inf\nolimits_{\sigma \in \mathit{Adv}_{\mathcal{M}}} \mathbb{E}^\sigma_{\mathcal{M},s}(X) \\ \mathbb{E}^{\max}_{\mathcal{M},s}(X) &\overset{\text{def}}{=} \sup\nolimits_{\sigma \in \mathit{Adv}_{\mathcal{M}}} \mathbb{E}^\sigma_{\mathcal{M},s}(X) \end{aligned}$$

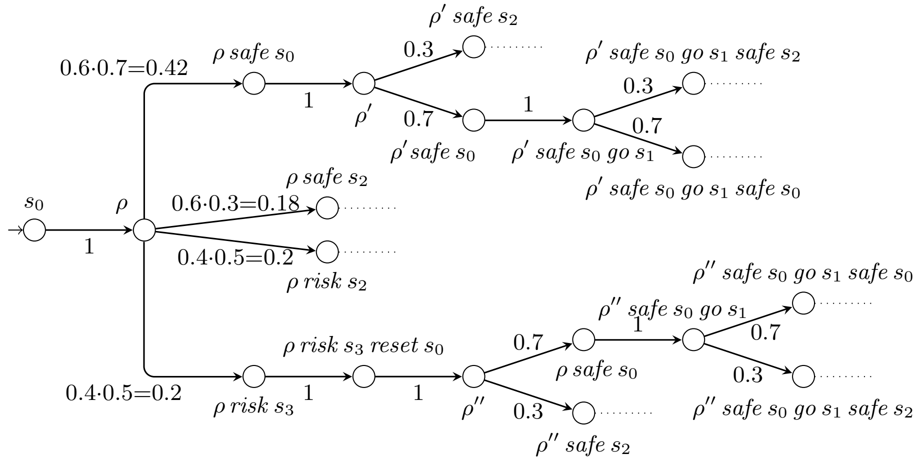


Fig. 3. Fragment of the induced DTMC $\mathcal{M}_\sigma$ (where $\rho = s_0\, go\, s_1$, $\rho' = \rho\, safe\, s_0\, go\, s_1$ and $\rho'' = \rho\, risk\, s_3\, reset\, s_0\, go\, s_1$)

When using an MDP to model and verify quantitative properties of a system, this equates to evaluating the *best-* or *worst-case* behaviour that can arise; for example, we may be interested in "the minimum probability of a message being delivered" or "the maximum expected time for a task to be completed".

Although not every subset of $IPath_{\mathcal{M}}$ is measurable, all the sets we will consider in this tutorial are measurable, so we can use the above notation freely without stressing the need of the measurability every time.

Example 3. Consider again the example MDP from Figure 2. Let σ be the adversary such that, for any finite path ρ:

$$\sigma(\rho) = \begin{cases} [go{\mapsto}1] & \text{if } last(\rho){=}s_0 \\ [risk{\mapsto}0.4, safe{\mapsto}0.6] & \text{if } \rho = s_0 \xrightarrow{go} s_1 \\ [safe{\mapsto}1] & \text{if } last(\rho){=}s_1 \text{ and } \rho \neq s_0 \xrightarrow{go} s_1 \\ [finish{\mapsto}1] & \text{if } last(\rho){=}s_2 \\ [reset{\mapsto}1] & \text{if } last(\rho){=}s_3\,. \end{cases}$$

Part of the induced DTMC $\mathcal{M}_\sigma$ is depicted in Figure 3. The adversary σ is neither memoryless, nor deterministic, but is finite-memory. More precisely, σ is defined by the tuple $(Q, \overline{q}, \sigma_u, \sigma_s)$ where $Q = \{q_0, q_1\}$, $\overline{q} = q_0$ and, for any $q \in Q$, $a \in \alpha_{\mathcal{M}}$ and $s \in S$:

$$\sigma_u(q,a,s) \;=\; \begin{cases} q_1 & \text{if } q{=}q_0 \text{ and } a \in \{risk, safe\} \\ q_0 & \text{if } q{=}q_0 \text{ and } a \notin \{risk, safe\} \\ q_1 & \text{otherwise} \end{cases}$$

and:

$$\sigma_s(q,s) \;=\; \begin{cases} [go{\mapsto}1] & \text{if } s{=}s_0 \\ [risk{\mapsto}0.4, safe{\mapsto}0.6] & \text{if } q{=}q_0 \text{ and } s{=}s_1 \\ [safe{\mapsto}1] & \text{if } q{=}q_1 \text{ and } s{=}s_1 \\ [finish{\mapsto}1] & \text{if } s{=}s_2 \\ [reset{\mapsto}1] & \text{if } s{=}s_3\,. \end{cases}$$

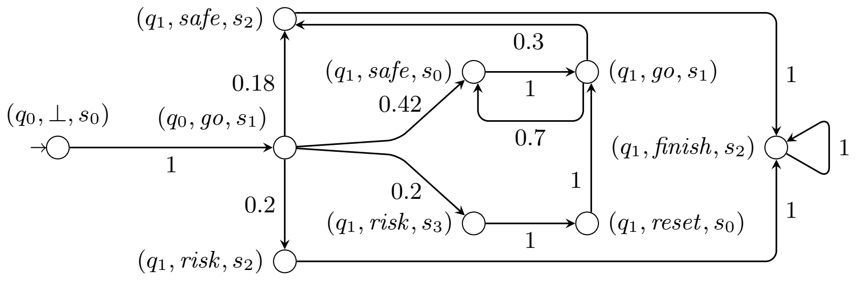


Fig. 4. The quotient DTMC $\mathcal{M}^q_\sigma$

The quotient DTMC $\mathcal{M}^q_\sigma$ is depicted in Figure 4.

Returning to the finite path $\rho = s_0 \xrightarrow{go} s_1 \xrightarrow{risk} s_3$ from Example 2, we have $Pr^\sigma_{\mathcal{M},s_0}(\{\pi \,|\, \rho \text{ is a prefix of } \pi\}) = (1{\cdot}1){\cdot}(0.4{\cdot}0.5) = 0.2$. ■

Related Models. We conclude this section by briefly surveying models that are closely related to MDPs and clarifying certain differences in terminology used elsewhere. Our definition of MDPs in this tutorial essentially coincides with the classical definitions (see e.g. [14,57,76]), although there are notational differences. Also commonly used in probabilistic verification is the model of (simple) *probabilistic automata* (PAs), due to Segala [80,81]. These permit multiple distributions labelled with the same action to be enabled from a state (i.e. $\delta_\mathcal{M}$ is a relation $\delta_\mathcal{M} \subseteq S \times \alpha_\mathcal{M} \times Dist(S)$), thus strictly generalising MDPs. This model is particularly well suited to compositional modelling and analysis of probabilistic systems, a topic that we will discuss further in Section 9. The names are sometimes used interchangeably, for example, the PRISM model checker [56] supports both PAs and MDPs, but refers to them simply as MDPs.

Confusingly, there is an alternative model called *probabilistic automata*, due to Rabin [78], which is also well known. From a syntactic point of view, these are essentially the same as MDPs, but are typically used in a language-theoretic setting, rather than for modelling and verification. An exception is [72], which uses Rabin's probabilistic automata to build a game-theoretic framework for verifying probabilistic programs. Another approach is to use "alternating" models, which distinguish between states that offer a probabilistic choice and those that offer a nondeterministic choice. Examples include the model of Hansson [53] and the concurrent Markov chains of [34,84]. We do not attempt a complete survey of MDP-like models here. See [48] for a classification scheme of such models, [82] for a thorough comparison and [80,1,7] for further references and discussion.

4 Probabilistic Reachability

In the remainder of this tutorial, we will introduce a variety of properties of MDPs and describe the corresponding methods to perform probabilistic model checking. We begin with the simple, yet fundamental, property of *probabilistic* *reachability.* This refers to the minimum or maximum probability, when starting from a given state s, of reaching a set of target states $T \subseteq S$. To formalise this, let $reach_s(T)$ be the set of all paths that start from state s and contain a state from T. More precisely, we have:

$$reach_s(T) \stackrel{\text{def}}{=} \{\pi \in IPath_{\mathcal{M},s} \mid \pi(i) \in T \text{ for some } i \in \mathbb{N}\}$$

and, when s is clear from the context, we will write $reach(T)$ instead of $reach_s(T)$. The measurability of $reach_s(T)$ follows from the fact that $reach_s(T) = \cup_{\rho \in I}\{\pi \in IPath_{\mathcal{M},s} \mid \pi \text{ has prefix } \rho\}$, where I is the (countable) set of all finite paths from s ending in T, and each element of this union is measurable. We then aim to compute one or both of the following probability bounds:

$$\begin{aligned} Pr^{\min}_{\mathcal{M},s}(reach_s(T)) &\stackrel{\text{def}}{=} \inf_{\sigma \in Adv_{\mathcal{M}}} Pr^{\sigma}_{\mathcal{M},s}(reach_s(T)) \\ Pr^{\max}_{\mathcal{M},s}(reach_s(T)) &\stackrel{\text{def}}{=} \sup_{\sigma \in Adv_{\mathcal{M}}} Pr^{\sigma}_{\mathcal{M},s}(reach_s(T)) \,. \end{aligned}$$

In the remainder of this section, we will first consider the special case of *qualitative* reachability, that is, finding those states for which the probability is either 0 or 1. Next, we consider the general *quantitative* case and discuss several different approaches that can be used to either compute or approximate minimum and maximum reachability probabilities. Finally, we describe how to generate adversaries which achieve the reachability probability of interest. Further details about many of the methods described in this section can be found in [76]. One important fact that we use is that there always exist *deterministic* and *memoryless* adversaries that achieve the minimum and maximum probabilities of reaching a target T.

4.1 Qualitative Reachability

In this section, we will present methods for finding the sets of states for which the minimum or maximum reachability probability is either 0 or 1. More precisely, we will be concerned with constructing the following sets of states:

$$\begin{aligned} S^0_{\min} &\stackrel{\text{def}}{=} \{s \in S \mid Pr^{\min}_s(reach_s(T)){=}0\} \\ S^1_{\min} &\stackrel{\text{def}}{=} \{s \in S \mid Pr^{\min}_s(reach_s(T)){=}1\} \\ S^0_{\max} &\stackrel{\text{def}}{=} \{s \in S \mid Pr^{\max}_s(reach_s(T)){=}0\} \\ S^1_{\max} &\stackrel{\text{def}}{=} \{s \in S \mid Pr^{\max}_s(reach_s(T)){=}1\} \,. \end{aligned}$$

The probability 0 cases are often *required* as input to the algorithms for quantitative reachability, while using both can reduce round-off errors and yield a speed-up in verification time. The gains are attributed to the fact that, to perform qualitative analysis, it is sufficient to know whether transitions are possible, not their precise probabilities. Hence, the analysis can be performed using *graph-based*, rather than numerical, computation.

Algorithms 1–4 give a formal description of how to compute the above sets; examples of executing these algorithms are presented in Section 4.2. For further details, see [18,1].

```
  Input: MDP $\mathcal{M} = (S, \overline{s}, \alpha_{\mathcal{M}}, \delta_{\mathcal{M}}, L)$, target set $T \subseteq S$
  Output: the set $S^0_{\min} = \{s \in S \mid Pr^{\min}_s(reach(T))=0\}$
1 $R := T$;
2 do
3     $R' := R$;
4     $R := R' \cup \{\, s \in S \mid \forall a \in A(s).\, (\, \exists s' \in R'.\, \delta_{\mathcal{M}}(s,a)(s')>0\,)\,\}$;
5 while $R \neq R'$ ;
6 return $S \setminus R$;
```

Algorithm 1. Computation of $S^0_{\min}$

```
  Input: MDP $\mathcal{M} = (S, \overline{s}, \alpha_{\mathcal{M}}, \delta_{\mathcal{M}}, L)$, set $S^0_{\min}$
  Output: the set $S^1_{\min} = \{s \in S \mid Pr^{\min}_s(reach(T))=1\}$
1 $R := S \setminus S^0_{\min}$;
2 do
3     $R' := R$;
4     $R := R' \setminus \{\, s \in R' \mid \exists a \in A(s).\, \big(\exists s' \in S.\, (\, \delta_{\mathcal{M}}(s,a)(s')>0 \wedge s' \notin R'\,)\big)\,\}$;
5 while $R \neq R'$ ;
6 return $R$;
```

Algorithm 2. Computation of $S^1_{\min}$

4.2 Quantitative Reachability

Before we introduce the actual algorithms for computing minimum and maximum reachability probabilities, we present the *Bellman equations* that describe these probabilities. It should be apparent that, if $x_s = Pr^{\min}_s(reach(T))$ for all $s \in S$, then the following equations are satisfied:

$$\begin{array}{ll} x_s = 1 & \text{if } s \in S^1_{\min} \\ x_s = 0 & \text{if } s \in S^0_{\min} \\ x_s = \min_{a \in A(s)} \sum_{s' \in S} \delta_{\mathcal{M}}(s,a)(s') \cdot x_{s'} & \text{otherwise.} \end{array}$$

When solving these equations, we in fact find the probability $Pr^{\min}_s(reach(T))$ *for all* states s of the MDP, rather than just a specific state of interest. From the results presented in [17,16] (since the problem of finding minimum reachability probabilities is a special case of the *stochastic shortest path problem*), the equations above have a *unique* solution. Furthermore, it is actually sufficient to just compute $S^0_{\min}$ and replace $S^1_{\min}$ with T in the above. Below, we will discuss various methods to solve these equations.

Similarly, if $x_s = Pr^{\max}_s(reach(T))$, then the following equations are satisfied:

$$\begin{array}{ll} x_s = 1 & \text{if } s \in S^1_{\max} \\ x_s = 0 & \text{if } s \in S^0_{\max} \\ x_s = \max_{a \in A(s)} \sum_{s' \in S} \delta_{\mathcal{M}}(s,a)(s') \cdot x_{s'} & \text{otherwise.} \end{array}$$

In this case, from the results of [17,1], it follows that the maximum reachability probabilities are the *least* solution to these equations. Again, like for the minimum case, it suffices to just compute $S^0_{\max}$ and replace $S^1_{\max}$ with T.

```
    **Input**: MDP $\mathcal{M} = (S, \overline{s}, \alpha_{\mathcal{M}}, \delta_{\mathcal{M}}, L)$, target set $T \subseteq S$
    **Output**: the set $S^0_{\max} = \{s \in S \mid Pr^{\max}_s(\mathit{reach}(T)){=}0\}$
1  $R := T$;
2  do
3      $R' := R$;
4      $R := R' \cup \{ s \in S \mid \exists a \in A(s). ( \exists s' \in R'. \delta_{\mathcal{M}}(s,a)(s'){>}0 ) \}$;
5  while $R \neq R'$ ;
6  return $S \setminus R$;
```

Algorithm 3. Computation of $S^0_{\max}$

```
     **Input**: MDP $\mathcal{M} = (S, \overline{s}, \alpha_{\mathcal{M}}, \delta_{\mathcal{M}}, L)$, target set $T \subseteq S$
     **Output**: $S^1_{\max} = \{s \in S \mid Pr^{\max}_s(\mathit{reach}(T)){=}1\}$
 1  $R := S$;
 2  do
 3      $R' := R$;
 4      $R := T$;
 5      do
 6          $R'' := R$;
 7          $R := R'' \cup \{s \in S \mid \exists a \in A(s).$
 8            $(\forall s' \in S. ( \delta_{\mathcal{M}}(s,a)(s'){>}0 \rightarrow s' \in R' ) ) \wedge ( \exists s' \in R''. \delta_{\mathcal{M}}(s,a)(s'){>}0 ) \}$;
 9      while $R \neq R''$ ;
10  while $R \neq R'$ ;
11  return $R$;
```

Algorithm 4. Computation of $S^1_{\max}$

Linear Programming. One approach to computing the minimum and maximum reachability probabilities is to construct and solve a *linear programming* (LP) problem. In the case of minimum probabilities $Pr^{\min}_s(\mathit{reach}(T))$, it has been demonstrated [17,35,1] that the following linear program:

maximise $\sum_{s \in S} x_s$ subject to the constraints:

$$
\begin{array}{ll}
x_s = 1 & \text{for all } s \in S^1_{\min} \\
x_s = 0 & \text{for all } s \in S^0_{\min} \\
x_s \leqslant \sum_{s' \in S} \delta_{\mathcal{M}}(s,a)(s') \cdot x_{s'} & \text{for all } s \notin S^1_{\min} \cup S^0_{\min} \text{ and } a \in A(s)
\end{array}
$$

has a unique solution satisfying $x_s = Pr^{\min}_s(\mathit{reach}(T))$.

Example 4. Consider again the example from Figure 2 and let us compute $Pr^{\min}_s(\mathit{reach}(\{s_2\}))$ for all $s \in S$. We first need to execute Algorithm 1, starting with $R = \{s_2\}$, and changing R to $\{s_1, s_2\}$ and to $\{s_0, s_1, s_2\}$ in two consecutive iterations of the do-while loop. This is a fixed point, so the returned set $S^0_{\min}$ is $S \backslash \{s_0, s_1, s_2\} = \{s_3\}$. Next, we execute Algorithm 2, starting with $R{=}\{s_0, s_1, s_2\}$ and then consecutively change R to $\{s_0, s_2\}$ and $\{s_2\}$, which is the fixed point, so $S^1_{\min} = \{s_2\}$. Using these sets, we obtain the linear program:

$$\begin{aligned}
&\text{maximise } x_{s_0}{+}x_{s_1}{+}x_{s_2}{+}x_{s_3} \text{ subject to:}\\
&x_{s_2} = 1\\
&x_{s_3} = 0\\
&x_{s_0} \leqslant x_{s_1}\\
&x_{s_1} \leqslant 0.5{\cdot}x_{s_2} + 0.5{\cdot}x_{s_3}\\
&x_{s_1} \leqslant 0.7{\cdot}x_{s_0} + 0.3{\cdot}x_{s_2}
\end{aligned}$$

which has the unique solution x_{s_0}=0.5, x_{s_1}=0.5, x_{s_2}=1 and x_{s_3}=0. Hence, the vector of values for $Pr_s^{\min}(\mathit{reach}(\{s_2\}))$ is $(0.5, 0.5, 1, 0)$. ■

In the case of maximum probabilities, the situation is similar [35,1], and we have the following the linear program:

$$\boxed{\begin{array}{ll}
\multicolumn{2}{l}{\text{minimise } \sum_{s\in S} x_s \text{ subject to the constraints:}}\\
x_s = 1 & \text{for all } s \in S^1_{\max}\\
x_s = 0 & \text{for all } s \in S^0_{\max}\\
x_s \geqslant \sum_{s'\in S} \delta(s,a)(s') \cdot x_{s'} & \text{for all } s \notin S^1_{\max} \cup S^0_{\max} \text{ and } a \in A(s)
\end{array}}$$

which yields a unique solution satisfying $x_s = Pr_s^{\max}(\mathit{reach}(T))$.

Example 5. We will illustrate the computation of maximum reachability probabilities using the MDP from Figure 2 and the target set $\{s_3\}$. We first run Algorithm 3, initialising the set R to $\{s_3\}$. After the first iteration of the do-while loop, we get R=$\{s_1, s_3\}$, and after the second we get R=$\{s_0, s_1, s_3\}$, which is the fixed point, so the returned set is $S^0_{\max}$=$\{s_2\}$. Then, we execute Algorithm 4. In the outer do-while loop (lines 2–10), we start with R'=S and R=$\{s_3\}$. The first execution of the inner loop (lines 5–9) yields R=$\{s_0, s_1, s_3\}$ and the second yields R=$\{s_3\}$. The latter is the fixed point, so we return $S^1_{\max}$=$\{s_3\}$. Setting up the linear program, we get:

$$\begin{aligned}
&\text{minimise } x_{s_0}{+}x_{s_1}{+}x_{s_2}{+}x_{s_3} \text{ subject to:}\\
&x_{s_3} = 1\\
&x_{s_2} = 0\\
&x_{s_0} \geqslant x_{s_1}\\
&x_{s_1} \geqslant 0.5{\cdot}x_{s_2} + 0.5{\cdot}x_{s_3}\\
&x_{s_1} \geqslant 0.7{\cdot}x_{s_0} + 0.3{\cdot}x_{s_2}
\end{aligned}$$

which has the unique solution x_{s_0}=0.5, x_{s_1}=0.5, x_{s_2}=0 and x_{s_3}=1, giving the vector of values $(0.5, 0.5, 0, 1)$ for $Pr_s^{\max}(\mathit{reach}(\{s_3\}))$. ■

The benefit of the linear programming approach is that it can be used to compute *exact* answers. The drawback, however, is that its scalability to large models is limited. Despite a wide range of LP solution methods being available, in practice models used in probabilistic verification become too large to solve in this way.

Value Iteration. An alternative method is *value iteration*, which offers better scalability than linear programming, but at the expense of accuracy. Instead of computing a precise solution to the set of linear equations for $Pr_s^{\min}(\mathit{reach}(T))$,

```
   Input: MDP $\mathcal{M} = (S, \overline{s}, \alpha_{\mathcal{M}}, \delta_{\mathcal{M}}, L)$, sets $S^0_{\min}, S^1_{\min}$ and convergence criterion $\varepsilon$
   Output: (approximation of) $Pr^{\min}_s(reach(T))$ for all $s \in S$
1  foreach $s \in S$ do  $x_s := \begin{cases} 1 \text{ if } s \in S^1_{\min} \\ 0 \text{ otherwise} \end{cases}$;
2  do
3      foreach $s \in S \backslash (S^0_{\min} \cup S^1_{\min})$ do
4          $x'_s := \min_{a \in A(s)} \sum_{s' \in S} \delta_{\mathcal{M}}(s,a)(s') \cdot x_{s'}$;
5      end
6      $\delta := \max_{s \in S} (x'_s - x_s)$;
7      foreach $s \in S \backslash (S^0_{\min} \cup S^1_{\min})$ do  $x_s := x'_s$;
8  while $\delta > \varepsilon$ ;
9  return $(x'_s)_{s \in S}$
```

Algorithm 5. Value iteration for $Pr^{\min}_s(reach(T))$

it computes the probability of reaching T within n steps. For large enough n, this yields a good enough approximation in practice.

Formally, we introduce variables x^n_s for $s \in S$ and $n \in \mathbb{N}$ and equations:

$$x^n_s = \begin{cases} 1 & \text{if } s \in S^1_{\min} \\ 0 & \text{if } s \in S^0_{\min} \\ 0 & \text{if } s \notin (S^1_{\min} \cup S^0_{\min}) \text{ and } n{=}0 \\ \min_{a \in A(s)} \sum_{s' \in S} \delta_{\mathcal{M}}(s,a)(s') \cdot x^{n-1}_{s'} & \text{otherwise.} \end{cases}$$

It can be shown [76,1,7] that $\lim_{n \to \infty} x^n_s = Pr^{\min}_s(reach(T))$. We can thus approximate $Pr^{\min}_s(reach(T))$ by computing x^n_s for sufficiently large n. Furthermore, we can compute the maximum probabilities $Pr^{\max}_s(reach(T))$ in near-identical fashion, by replacing "min" with "max" in the above.

Typically, a suitable value of n is not decided in advance, but rather determined on-the-fly, based on the convergence of the values x^n_s. A simple but effective scheme is to terminate the computation when $\max_{s \in S}(x^n_s - x^{n-1}_s)$ drops below a specified threshold ε. In cases where the probability values are very small, the maximum *relative* difference, i.e. $\max_{s \in S}((x^n_s - x^{n-1}_s)/x^{n-1}_s)$ may be a more reliable criterion. It is important to stress, however, that these tests do *not* guarantee that the resulting values are within ε of the true answer. In theory, it is possible to make certain guarantees on the precision obtained, based on the denominators of the (rational) transition probabilities [27]. However, it is unclear whether these are practically applicable.

An illustration of how value iteration can be implemented is given in Algorithm 5. In practice, there is no need to store all vectors $\mathbf{x}^n$; just two ($\mathbf{x}$ and $\mathbf{x}'$ in the algorithm) are required.

Example 6. To illustrate value iteration, we will slightly modify the running example from Figure 2: let us suppose that the action *reset* is not available in s_3, and that the action *safe* is not completely reliable, but results in a failure with probability 0.1 and leads to s_0 only with probability 0.6. The modified MDP is depicted in Figure 5. Suppose we want to compute $Pr^{\max}_{s_0}(reach(\{s_2\}))$. By

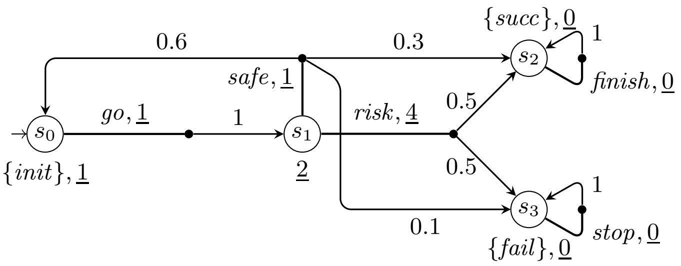


Fig. 5. A modified version of the running example from Figure 2

executing Algorithms 3 and 4, we obtain $S^0_{\max}=\{s_3\}$ and $S^1_{\max}=\{s_2\}$, yielding the following equations for value iteration:

$$\begin{array}{lll}
x^n_{s_2} &= 1 & \text{for } n\geqslant 0 \\
x^n_{s_3} &= 0 & \text{for } n\geqslant 0 \\
x^0_{s_i} &= 0 & \text{for } i \in \{0,1\} \\
x^n_{s_0} &= x^{n-1}_{s_1} & \text{for } n>0 \\
x^n_{s_1} &= \max\{0.6{\cdot}x^{n-1}_{s_0} + 0.1{\cdot}x^{n-1}_{s_3} + 0.3{\cdot}x^{n-1}_{s_2}, 0.5{\cdot}x^{n-1}_{s_2} + 0.5{\cdot}x^{n-1}_{s_3}\} & \text{for } n>0
\end{array}$$

Below, are the vectors $\mathbf{x}^n = (x^n_{s_0}, x^n_{s_1}, x^n_{s_2}, x^n_{s_3})$, shown up to a precision of 5 decimal places, for increasing n, terminating with ε=0.001.

$$\begin{array}{ll}
\mathbf{x}^0 = (0.0,\ 0.0,\ 1.0, 0.0) & \mathbf{x}^7 = (0.66,\ 0.696,\ 1.0, 0.0) \\
\mathbf{x}^1 = (0.0,\ 0.5,\ 1.0, 0.0) & \mathbf{x}^8 = (0.696,\ 0.696,\ 1.0, 0.0) \\
\mathbf{x}^2 = (0.5,\ 0.5,\ 1.0, 0.0) & \mathbf{x}^9 = (0.696,\ 0.7176,\ 1.0, 0.0) \\
\mathbf{x}^3 = (0.5,\ 0.6,\ 1.0, 0.0) & \mathbf{x}^{10} = (0.7176,\ 0.7176,\ 1.0, 0.0) \\
\mathbf{x}^4 = (0.6,\ 0.6,\ 1.0, 0.0) & \cdots \\
\mathbf{x}^5 = (0.6,\ 0.66, 1.0, 0.0) & \mathbf{x}^{22} = (0.74849, 0.74849, 1.0, 0.0) \\
\mathbf{x}^6 = (0.66, 0.66, 1.0, 0.0) & \mathbf{x}^{23} = (0.74849, 0.74909, 1.0, 0.0)
\end{array}$$

The exact values for $Pr^{\max}_s(reach(\{s_2\}))$ are $(0.75, 0.75, 1, 0)$, which differ from $\mathbf{x}^{23}$ by up to 0.00151 (for state s_0). ■

Gauss-Seidel Value Iteration. Several variants of value iteration exist that improve its efficiency. One such variant is *Gauss-Seidel* value iteration. Intuitively, this method exhibits faster convergence by using the most up-to-date probability values available for each state within each iteration of the computation. This has the added benefit that only a single vector of probabilities needs to be stored, since new values are written directly to the vector. Algorithm 6 shows the algorithm for the case of minimum reachability probabilities. Notice that it uses only a single vector $\mathbf{x}$.

Policy Iteration. An alternative class of algorithms for computing reachability probabilities is *policy iteration* (recall that "policy" is alternative terminology for "adversary"). Whereas value iteration steps through vectors of values, policy iteration generates a sequence of adversaries. We start with an arbitrary, deterministic and memoryless adversary, and then repeatedly construct an improved (deterministic and memoryless) adversary by computing the corresponding probabilities and changing the actions taken so that the probability of reaching T is

```
   Input: MDP $\mathcal{M} = (S, \overline{s}, \alpha_{\mathcal{M}}, \delta_{\mathcal{M}}, L)$, sets $S^0_{\min}, S^1_{\min}$ and convergence criterion $\varepsilon$
   Output: (approximation of) $Pr^{\min}_s(reach(T))$ for all $s \in S$
 1 foreach $s \in S$ do $x_s := \begin{cases} 1 \text{ if } s \in S^1_{\min} \\ 0 \text{ otherwise} \end{cases}$;
 2 do
 3     $\delta := 0$;
 4     foreach $s \in S \backslash (S^0_{\min} \cup S^1_{\min})$ do
 5         $x_{new} := \min_{a \in A(s)} \sum_{s' \in S} \delta_{\mathcal{M}}(s,a)(s') \cdot x_{s'}$;
 6         $\delta := \max(\delta, x_{new} - x_s)$;
 7         $x_s := x_{new}$;
 8     end
 9 while $\delta > \varepsilon$ ;
10 return $(x_s)_{s \in S}$
```

Algorithm 6. Gauss-Seidel value iteration for $Pr^{\min}_s(reach(T))$

decreased or increased (depending on whether minimum or maximum probabilities are being computed). The existence of deterministic and memoryless adversaries exhibiting minimum and maximum reachability probabilities, together with properties of the reachability problem, implies that this method will return the correct result (assuming it terminates). Termination is guaranteed by the fact that there are only finitely many such adversaries.

Algorithm 7 describes policy iteration for computing $Pr^{\min}_s(reach(T))$; the case for maximum values is similar. Notice that, since the adversary is both deterministic and memoryless, we can describe it simply as a mapping σ from states to actions. Computing the probabilities $Pr^{\sigma}_s(reach(T))$ for each adversary σ is relatively straightforward and is done by computing reachability probabilities for the corresponding quotient DTMC $\mathcal{M}^q_\sigma$. Since $\mathcal{M}^q_\sigma$ is finite-state, this can be done either by treating it as a (finite-state) MDP and using the other methods described in this section, or by solving a linear equation system [67]. We remark also that, to ensure termination of the algorithm, the action assigned to $\sigma(s)$ when improving the policy should only be changed when there is a strict improvement in the probability for s.

```
  Input: MDP $\mathcal{M} = (S, \overline{s}, \alpha_{\mathcal{M}}, \delta_{\mathcal{M}}, L)$, target set $T \subseteq S$
  Output: $Pr^{\min}_s(reach(T))$ for all $s \in S$
1 Pick arbitrary adversary $\sigma$;
2 do
3     Compute $p_s := Pr^{\sigma}_s(reach(T))$ for all $s \in S$;
4     foreach $s \in S$ do $\sigma(s) := \arg\min_{a \in A(s)} \sum_{s' \in S} \delta_{\mathcal{M}}(s,a)(s') \cdot p_{s'}$
5 while $\sigma$ has changed ;
6 return $(p_s)_{s \in S}$
```

Algorithm 7. Policy iteration for $Pr^{\min}_s(reach(T))$

Example 7. To demonstrate the policy iteration method, let us modify the MDP from Figure 2 by adding a self-loop on s_0 labelled with a new action *wait* (i.e. $\delta_{\mathcal{M}}(s_0, \mathit{wait}) = [s_0 \mapsto 1]$). Note that there are $2^3{=}8$ different deterministic and memoryless adversaries in the new MDP. Let us maximise the probability of reaching $\{s_2\}$. We start with the adversary σ that picks *wait*, *safe* and *stop* in s_0, s_1 and s_3, respectively. The vector of probabilities of reaching the state s_2 under σ is $(0, 0.3, 1, 0)$. We then change the decision of σ in s_0 to *go*, and the decision in s_1 to *risk*. Recomputing the values p_s, we get $(0.5, 0.5, 1, 0)$ and subsequently change the decision of σ in s_1 back to *safe* and in s_3 to *reset*. Computing the values of p_s then yields $(1, 1, 1, 1)$, which cannot be further improved, so these probabilities are returned. ■

Method Comparison. To give an idea of the relative performance of the computation methods described in this section, we provide a small set of results, using models from the PRISM benchmark suite [89]. Table 1 shows results for 8 model checking problems on 6 different models. The models, and associated parameters, are: *consensus* ($N{=}4, K{=}16$), *firewire_dl* (*delay*=36, *deadline*=800), *csma* ($N{=}3, K{=}4$), *wlan* (*BOFF*=5, *COL*=6), *zeroconf* ($N{=}1000, K{=}8$, *reset*=*f*), *zeroconf_dl* ($N{=}1000, K{=}2$, *reset*=*f*, *bound*=30); see [89] for further details.

We show the model sizes (number of states) and, for each of three methods (value iteration, Gauss-Seidel, policy iteration), the number of iterations needed and the total solution time (in seconds), running on a 2.53 GHz machine with 8 GB RAM. For policy iteration, we use Gauss-Seidel to analyse each adversary and, for all iterative methods, we terminate when the maximum absolute difference is below $\varepsilon{=}10^{-6}$. Results are omitted for linear programming since this approach does not scale to these model sizes. We see that Gauss-Seidel is always faster than standard value iteration and often gives a significant speed-up, thanks to its faster convergence. It is likely that further gains could be made by re-ordering the state space. Policy iteration, in most cases, needs to examine a relatively small number of adversaries. However, it does not (on this benchmark set, at least) offer any improvement over Gauss-Seidel value iteration and on some models can be considerably slower than standard value iteration.

4.3 Adversary Generation

As stated earlier in this section, for MDP $\mathcal{M}$ and target set T, there are always *deterministic* and *memoryless* adversaries, say $\sigma^{\min}$ and $\sigma^{\max}$, exhibiting the minimum and maximum probabilities of reaching T, respectively. So far, we have described how to compute the values of these probability bounds. Here we discuss how to generate adversaries $\sigma^{\min}$ and $\sigma^{\max}$ that achieve these bounds. As for policy iteration discussed earlier, since the adversaries are deterministic and memoryless, we can describe them as a mapping from states to actions.

For the case of minimum probabilities, this is straightforward. Regardless of the method used for computation of the minimum probabilities $Pr_s^{\min}(\mathit{reach}(T))$, we define, for each state $s \in S$:

$$\sigma^{\min}(s) \stackrel{\text{def}}{=} \arg\min_{a \in A(s)} \big(\textstyle\sum_{s' \in S} \delta_{\mathcal{M}}(s,a)(s') \cdot Pr_{s'}^{\min}(\mathit{reach}(T)) \big) .$$

Table 1. Comparison of the methods for computing reachability probabilities

Model	Property	Size	Value iter.		Gauss-Seidel		Policy iteration		
		(states)	Iter.s	Time	Iter.s	Time	Adv.s	Iter.s	Time
consensus	*c2*	166,016	70,681	368.8	35,403	**190.1**	6	35,687	196.1
consensus	*disagree*	166,016	90,861	637.6	45,432	**336.5**	52	83,775	533.3
firewire_dl	*deadline*	530,965	621	5.74	614	**5.67**	1	614	5.69
csma	*all_before*	1,460,287	118	3.53	87	**2.44**	2	161	5.68
csma	*some_before*	1,460,287	59	0.25	45	**0.14**	2	76	0.58
wlan	*collisions*	1,591,710	825	2.34	323	**0.97**	4	788	2.58
zeroconf	*correct*	1,870,338	345	16.6	259	**14.4**	4	581	24.8
zeroconf_dl	*deadline*	666,378	122	1.09	81	**0.76**	18	758	6.86

For the case of maximum probabilities, more care is required [1]. There are several solutions, depending on the probability computation method used. If policy iteration was applied, for example, the process is trivial since adversaries are explicitly constructed and solved during the algorithm's execution. We will now demonstrate how to adapt the value iteration algorithm to compute an optimal adversary as it proceeds. The idea is essentially the same as for minimum probabilities above, but we perform this at every step of the computation and, crucially, only update the adversary for a state s if the probability is strictly better in that state. The adapted algorithm is shown in Algorithm 8. An additional caveat of this version of the algorithm is that we skip the (optional) computation of $S^1_{\max}$ in order to correctly determine optimal choices for those states. For the states in $S^0_{\max}$, by definition the reachability probability is 0 for all adversaries, and hence we can choose arbitrary actions in these states.

5 Reward-Based Properties

Reward structures are a powerful mechanism for reasoning about various quantitative properties of models. As mentioned earlier, reward structures can be used to model system characteristics such as power consumption, the cost or time to execute a task and the size of a queue. In this tutorial, we discuss *expected reward* properties, focussing on two particular classes: *instantaneous reward* and *cumulative reward*. Instantaneous reward allows reasoning about the expected reward associated with the state of an MDP after a particular number of steps. Cumulative reward is useful in situations where we are interested in the sum of rewards accumulated up to a certain point. There are also various other reward-based properties for MDPs, of which two of the most prominent are:

- *Discounted reward*, in which reward is accumulated step by step, but the reward gained in the nth step is multiplied by a factor λ^n for some $\lambda < 1$, thus giving preference to rewards gained earlier in time;
- *Expected long-run average reward*, in which the average reward gained per state or transition is considered.

```
Input: MDP $\mathcal{M}$, target $T$, set $S^0_{\max}$ and convergence criterion $\varepsilon$
Output: (approximation of) $Pr^{\max}_s(reach(T))$ for all $s \in S$, optimal adversary
1  foreach $s \in S \backslash (S^0_{\max} \cup T)$ do
2      $x_s := \begin{cases} 1 \text{ if } s \in T \\ 0 \text{ otherwise} \end{cases}$;
3      $\sigma^{\max}(s) := \perp$;
4  end
5  do
6      foreach $s \in S \backslash (S^0_{\max} \cup T)$ do
7          $x'_s := \max_{a \in A(s)} \sum_{s' \in S} \delta_{\mathcal{M}}(s,a)(s') \cdot x_{s'}$;
8          if $\sigma^{\max}(s) = \perp$ or $x'_s > x_s$ then
9              $\sigma^{\max}(s) := \arg\max_{a \in A(s)} \sum_{s' \in S} \delta_{\mathcal{M}}(s,a)(s') \cdot x_{s'}$';
10         end
11         $\delta := \max_{s \in S} (x'_s - x_s)$;
12         foreach $s \in S \backslash (S^0_{\max} \cup T)$ do  $x_s := x'_s$;
13     end
14 while $\delta > \varepsilon$ ;
15 return $(x'_s)_{s \in S}, \sigma^{\max}$
```

Algorithm 8. Value iteration/adversary generation for $Pr^{\max}_s(reach(T))$

The reader is refered to [76] for a more comprehensive review of these and other such properties.

5.1 Instantaneous Rewards

One of the simplest MDP reward properties is *instantaneous reward*, which is defined as the expected reward of the state entered after k steps, for a given reward structure and step $k \in \mathbb{N}$. For example, if the MDP models a system equipped with a queue and the reward structure assigns the current queue size to each state, then instantaneous reward properties allow us to formalise questions such as "what is the maximum expected size of the queue after 200 steps?".

Formally, given an MDP $\mathcal{M} = (S, \overline{s}, \alpha_{\mathcal{M}}, \delta_{\mathcal{M}}, L)$, state reward function r_{state} for $\mathcal{M}$, state $s \in S$ and step k, we define a random variable $I^{=k}_{r_{state}} : IPath_s \to \mathbb{R}_{\geqslant 0}$ such that $I^{=k}_{r_{state}}(\pi) \overset{\text{def}}{=} r_{state}(\pi(k))$ for all infinite paths $\pi \in IPath_s$. The value of interest is then either $\mathbb{E}^{\min}_s(I^{=k}_{r_{state}})$ or $\mathbb{E}^{\max}_s(I^{=k}_{r_{state}})$, i.e. either the minimum or maximum expected reward at step k when starting from state s.

It is easy to see that memoryless adversaries do not suffice for minimising or maximising instantaneous reward. For example, consider the MDP from Figure 6 and the value of $\mathbb{E}^{\max}_{s_0}(I^{=3}_{r_{state}})$. The optimal behaviour is to take self-loop b_0 twice and then the action a_0, which yields expected instantaneous reward of 2, which no memoryless adversary can achieve. Intuitively, the adversary may need to "wait" in some states until the time comes to take a step towards states in which the reward is large (or small if we consider the minimising case).

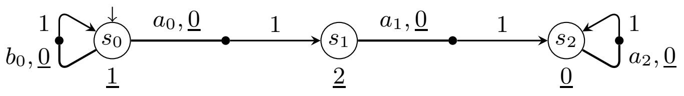


Fig. 6. An example MDP for which optimal memoryless adversaries do not exist

The values $\mathbb{E}^{\min}_s(I^{=k}_{r_{state}})$ are computed through the following equations, which exploit the relation between instantaneous reward in the ℓth and $(\ell{-}1)$th steps:

$$x^\ell_s = \begin{cases} r_{state}(s) & \text{if } \ell{=}0 \\ \min_{a\in A(s)} \big(\sum_{s'\in S} \delta_{\mathcal{M}}(s,a)(s')\cdot x^{\ell-1}_{s'} \big) & \text{otherwise} \end{cases}$$

We set $\mathbb{E}^{\min}_s(I^{=k}_{r_{state}})$ equal to x^k_s. It is also straightforward to extend this to compute an optimal adversary $\sigma^{\min}$ on-the-fly by remembering the action:

$$a^\ell_s = \arg\min_{a\in A(s)} \big(\textstyle\sum_{s'\in S} \delta_{\mathcal{M}}(s,a)(s')\cdot x^{\ell-1}_{s'} \big)$$

for all $1{\leqslant}\ell{\leqslant}k$ and $s \in S$, and setting $\sigma^{\min}(\rho) = [a_\rho{\mapsto}1]$ where $a_\rho = a^{k-|\rho|}_{last(\rho)}$ for all $\rho \in \mathit{FPath}$ such that $|\rho|{\leqslant}k{-}1$. The choices for paths longer than $k{-}1$ can be arbitrary as these do not influence the expected reward.

The equations for computing $\mathbb{E}^{\max}_s(I^{=k}_{r_{state}})$ and $\sigma^{\max}$ can be obtained by replacing "min" with "max" in those above for $\mathbb{E}^{\min}_s(I^{=k}_{r_{state}})$.

Example 8. Let us compute the maximum instantaneous reward after 4 steps in the MDP from Figure 2. This amounts to finding the solution to the equations:

$$\begin{aligned}
x^0_{s_0} &= 1 \\
x^0_{s_1} &= 2 \\
x^0_{s_2} &= 0 \\
x^0_{s_3} &= 0 \\
x^\ell_{s_0} &= x^{\ell-1}_{s_1} \\
x^\ell_{s_1} &= \max\{0.7{\cdot}x^{\ell-1}_{s_0} + 0.3{\cdot}x^{\ell-1}_{s_2}, 0.5{\cdot}x^{\ell-1}_{s_2} + 0.5{\cdot}x^{\ell-1}_{s_3}\} \\
x^\ell_{s_2} &= x^{\ell-1}_{s_2} \\
x^\ell_{s_3} &= \max\{x^{\ell-1}_{s_3}, x^{\ell-1}_{s_0}\}
\end{aligned}$$

for $1{\leqslant}\ell{\leqslant}4$. The following are the values $\mathbf{x}^i = (x^i_{s_0}, x^i_{s_1}, x^i_{s_2}, x^i_{s_3})$:

$$\begin{aligned}
\mathbf{x}^1 &= (\ \ 2, 0.7, 0, 1) \\
\mathbf{x}^2 &= (\ 0.7, 1.4, 0, 2) \\
\mathbf{x}^3 &= (\ 1.4, \ \ 1, 0, 2) \\
\mathbf{x}^4 &= (\ \ 1, \ \ 1, 0, 2)
\end{aligned}$$

So, e.g., $\mathbb{E}^{\max}_{s_0}(I^{=4}_{r_{state}}) = 1$. The associated optimal adversary $\sigma^{\max}$ satisfies:

– if $last(\rho){=}s_1$, then $\sigma^{\max}(\rho){=}[\mathit{risk}{\mapsto}1]$ when $|\rho|{\leqslant}1$ and $[\mathit{safe}{\mapsto}1]$ otherwise;
– if $last(\rho){=}s_3$, then $\sigma^{\max}(\rho){=}[\mathit{stop}{\mapsto}1]$ when $|\rho|{\leqslant}1$ and $[\mathit{reset}{\mapsto}1]$ otherwise. ■

5.2 Step-Bounded Cumulative Reward

Rather than computing the expected reward gained *at* the kth step, it may be useful to compute the expected reward *accumulated up to* the kth step. Formally, given an MDP $\mathcal{M}{=}(S,\overline{s},\alpha_{\mathcal{M}},\delta_{\mathcal{M}},L)$, reward structure $r{=}(r_{state},r_{action})$, state s and step bound k, we define the random variable $C_r^{\leqslant k} : \mathit{IPath}_s \to \mathbb{R}_{\geqslant 0}$ where:

$$C_r^{\leqslant k}(\pi) \stackrel{\text{def}}{=} \textstyle\sum_{i=0}^{k-1} \big(r_{state}(s_i) + r_{action}(s_i,a_i) \big)$$

for all infinite paths $\pi = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \cdots \in \mathit{IPath}_s$ and consider either $\mathbb{E}_s^{\min}(C_r^{\leqslant k})$ or $\mathbb{E}_s^{\max}(C_r^{\leqslant k})$. For example, if the MDP models a system in which energy is consumed at each step, and the reward structure assigns energy values to actions, then step-bounded cumulative reward can be used to reason about properties such as "the expected energy consumption of the system within the first 1000 time-units of operation".

As for the previous case, there may not exist a memoryless optimal adversary. For example, consider $\mathbb{E}_{s_0}^{\min}(C_r^{\leqslant 3})$ for the MDP from Figure 6. The optimal behaviour is to take the self-loop b_0 once and then the actions a_0 and a_1, which yields cumulated reward 5. This is not achievable by any memoryless adversary.

The value of $\mathbb{E}_s^{\min}(C_r^{\leqslant k})$ is computed in a similar way to instantaneous rewards through the following equations:

$$x_s^\ell = \begin{cases} 0 & \text{if } \ell{=}0 \\ r_{state}(s) + \min_{a\in A(s)} \big(r_{action}(s,a) + \sum_{s'\in S} \delta_{\mathcal{M}}(s,a)(s')\cdot x_{s'}^{\ell-1} \big) & \text{otherwise} \end{cases}$$

Like for the instantaneous case, an optimal adversary can be constructed on-the-fly by remembering the action:

$$a_s^\ell = \arg\min_{a\in A(s)} \big(r_{action}(s,a) + \textstyle\sum_{s'\in S} \delta_{\mathcal{M}}(s,a)(s')\cdot x_{s'}^{\ell-1} \big)$$

for all $1{\leqslant}\ell{\leqslant}k$ and $s \in S$, and setting $\sigma(\rho) = [a_\rho{\mapsto}1]$ where $a_\rho = a_{last(\rho)}^{k-|\rho|}$ for all $\rho \in \mathit{FPath}$ such that $|\rho|{\leqslant}k{-}1$. The choices for paths longer than $k{-}1$ can be arbitrary as these do not influence the expected reward.

When considering maximum rewards, the corresponding equations can be obtained by replacing "min" with "max" in those above.

Example 9. Let us compute the maximum expected reward accumulated within 4 steps in the MDP from Figure 2. In order to do this, we need to solve the following equations (cf. Example 8):

$$\begin{aligned}
x_{s_0}^0 &= 0 \\
x_{s_1}^0 &= 0 \\
x_{s_2}^0 &= 0 \\
x_{s_3}^0 &= 0 \\
x_{s_0}^\ell &= 1 + 1 + x_{s_1}^{\ell-1} \\
x_{s_1}^\ell &= 2 + \max\{1 + 0.7\cdot x_{s_0}^{\ell-1} + 0.3\cdot x_{s_2}^{\ell-1}, 4 + 0.5\cdot x_{s_2}^{\ell-1} + 0.5\cdot x_{s_3}^{\ell-1}\} \\
x_{s_2}^\ell &= x_{s_2}^{\ell-1} \\
x_{s_3}^\ell &= \max\{x_{s_3}^{\ell-1}, 5 + x_{s_0}^{\ell-1}\}
\end{aligned}$$

for $1 \leqslant \ell \leqslant 4$. The following are the values $\mathbf{x}^i = (x^i_{s_0}, x^i_{s_1}, x^i_{s_2}, x^i_{s_3})$:

$$\begin{aligned}
\mathbf{x}^1 &= \ \ \ (2, \ \ \ 6, 0, \ \ \ 5) \\
\mathbf{x}^2 &= \ \ \ (8, \ 8.5, 0, \ \ \ 7) \\
\mathbf{x}^3 &= (10.5, \ \ 9.5, 0, \ \ 13) \\
\mathbf{x}^4 &= (11.5, 12.5, 0, 15.5)
\end{aligned}$$

So, e.g., $\mathbb{E}^{\max}_{s_0}(C^{\leqslant 4}_r) = 11.5$. The computed optimal adversary $\sigma^{\max}$ is in fact memoryless and satisfies $\sigma^{\max}(s_1) = [\mathit{risk} \mapsto 1]$ and $\sigma^{\max}(s_3) = [\mathit{reset} \mapsto 1]$. ■

5.3 Cumulative Reward to Reach a Target

Sometimes it is more useful to consider the cumulative reward gained before some set of target states is reached, rather than within a time bound. This could be used, for example, to compute "the expected cost of completing a task" or "the total expected energy consumption during a system's lifetime".

Let $\mathcal{M} = (S, \overline{s}, \alpha_{\mathcal{M}}, \delta_{\mathcal{M}}, L)$ be an MDP, $r = (r_{state}, r_{action})$ a reward structure, $s \in S$ and $T \subseteq S$ a set of target states. We aim to compute $\mathbb{E}^{\min}_s(F^T_r)$ or $\mathbb{E}^{\max}_s(F^T_r)$ where $F^T_r : \mathit{IPath}_s \to \mathbb{R}_{\geqslant 0}$ is the random variable such that, for any path $\pi = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \cdots \in \mathit{IPath}_s$:

$$F^T_r(\pi) \stackrel{\text{def}}{=} \begin{cases} \infty & \text{if } s_i \notin T \text{ for all } i \in \mathbb{N} \\ \sum_{i=0}^{k^T_\pi - 1} \left(r_{state}(s_i) + r_{action}(s_i, a_i) \right) & \text{otherwise} \end{cases}$$

where $k^T_\pi = \min\{k \mid s_k \in T\}$. As for probabilistic reachability, there are always *deterministic* and *memoryless* adversaries exhibiting the minimum and maximum expected cumulative reward of reaching a target T.

Let us first consider the case $\mathbb{E}^{\min}_s(F^T_r)$. By definition, the value of $\mathbb{E}^{\min}_s(F^T_r)$ is infinite if and only if, for all adversaries, when starting in state s, the probability of reaching T is strictly less than 1. Using the methods presented in Section 4, we can compute $S^1_{\max}$, i.e. the set of states for which the probability of reaching T equals 1 for some adversary. Hence, $\mathbb{E}^{\min}_s(F^T_r)$ is infinite if and only if $s \notin S^1_{\max}$, and it remains to compute the values for the states in $s \in S^1_{\max}$. Let $C^T_r : \mathit{IPath}_s \to \mathbb{R}_{\geqslant 0}$ be the random variable such that for any path $\pi = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \cdots \in \mathit{IPath}_s$:

$$C^T_r(\pi) \stackrel{\text{def}}{=} \sum_{i=0}^{l^T_\pi - 1} \left(r_{state}(s_i) + r_{action}(s_i, a_i) \right)$$

where $l^T_\pi = \infty$ if $s_i \notin T$ for all $i \in \mathbb{N}$ and equals $\min\{k \mid s_k \in T\}$ otherwise. Now, using [1,17], if there exists a *proper* adversary that achieves the minimum expected value of C^T_r, i.e. an adversary σ such that:

$$\mathit{Pr}^\sigma_s(\mathit{reach}(T)) = 1 \ \text{ and } \ \mathbb{E}^\sigma_s(C^T_r) = \mathbb{E}^{\min}_s(C^T_r) \quad \text{for all } s \in S^1_{\max},$$

then the values $\mathbb{E}^{\min}_s(F^T_r)$ are the unique solution to the following equations:

$$x_s = \begin{cases} 0 & \text{if } s \in T \\ r_{state}(s) + \min_{a \in A(s)} \left(r_{action}(s, a) + \sum_{s' \in S} \delta_{\mathcal{M}}(s, a)(s') \cdot x_{s'} \right) & \text{otherwise.} \end{cases}$$

As for probabilistic reachability in Section 4, techniques such as linear programming, value iteration or policy iteration can be used to compute the value. However, there still remains the problem of checking for the existence of such a proper adversary or dealing with the case when no such proper adversary exists. The solution is to use the techniques presented in [1], and perform a transformation of the MDP, by essentially removing end components with only zero rewards, to guarantee that such a proper adversary exists.

For the maximum case, by definition, the value $\mathbb{E}^{\max}_s(F^T_r)$ is infinite if and only if there is an adversary under which T is reached from s with probability strictly less than 1. Again, using the methods presented in Section 4, we can compute $S^1_{\min}$, i.e. the set of states for which no such adversary exists. Hence, $S^1_{\min}$ identifies precisely those states for which the maximum expected reward is finite. For such states s, the values $\mathbb{E}^{\max}_s(F^T_r)$ satisfy the follow equations:

$$x_s = \begin{cases} 0 & \text{if } s \in T \\ r_{state}(s) + \max_{a \in A(s)} \big(r_{action}(s,a) + \sum_{s' \in S} \delta_{\mathcal{M}}(s,a)(s') \cdot x_{s'} \big) & \text{otherwise} \end{cases}$$

and are in this case the least solution [1]. We can again use techniques such as those described in Section 4 to find this solution.

Example 10. Let us consider the MDP from Figure 5, and compute the minimum expected reward to reach $\{s_2, s_3\}$. Since $S^1_{\max} = S$, we obtain the equations:

$$\begin{aligned} x_{s_2} &= 0 \\ x_{s_3} &= 0 \\ x_{s_0} &= 1 + 1 + x_{s_1} \\ x_{s_1} &= 2 + \min\{4 + 0.5 \cdot x_{s_2} + 0.5 \cdot x_{s_3}, 1 + 0.3 \cdot x_{s_2} + 0.1 \cdot x_{s_3} + 0.6 \cdot x_{s_0}\} \end{aligned}$$

The unique solution is the vector $(8, 6, 0, 0)$ and, e.g., $\mathbb{E}^{\min}_{s_0}(F^{\{s_2,s_3\}}_r) = 8$. ■

A careful reader may have noticed that both properties considered earlier in this section, i.e. instantaneous reward at the kth step and cumulative reward up to the kth step, can be reduced to the cumulative reward to reach a target set. More precisely, for an MDP $\mathcal{M} = (S, \overline{s}, \alpha_{\mathcal{M}}, \delta_{\mathcal{M}}, L)$ and reward structure $r = (r_{state}, r_{action})$, we can construct an MDP $\mathcal{M}' = (S', \overline{s}', \delta_{\mathcal{M}'}, L')$, reward structure $r' = (r'_{state}, r'_{action})$ and target T' where in both cases:

- $S' = S \times \{0, \ldots, k{+}1\}$ and $\overline{s}' = (\overline{s}, 0)$;
- for any $s, s' \in S$, $a \in \alpha_{\mathcal{M}}$ and $i \leqslant k$ we have $\delta_{\mathcal{M}'}((s,i),a)((s',i{+}1)) = \delta_{\mathcal{M}}(s,a)(s')$ and $\delta_{\mathcal{M}'}((s,k{+}1),a)((s',k{+}1)) = \delta_{\mathcal{M}}(s,a)(s')$.

In the instantaneous case, $T' = S \times \{k{+}1\}$ and, for any $s \in S$, $0 \leqslant i \leqslant k{+}1$ and $a \in \alpha_{\mathcal{M}}$, we set $r'_{state}(s,i)$ to $r_{state}(s)$ if $i{=}k$ and to 0 otherwise, and $r'_{action}((s,i),a){=}0$. We then have that $\mathbb{E}^{\min}_{(s,0)}(F^{T'}_{r'})$ (respectively $\mathbb{E}^{\max}_{(s,0)}(F^{T'}_{r'})$) in $\mathcal{M}'$ equals $\mathbb{E}^{\min}_s(I^{=k}_{r_{state}})$ (respectively $\mathbb{E}^{\max}_s(I^{=k}_{r_{state}})$) in $\mathcal{M}$.

For cumulative rewards, $T' = S \times \{k\}$, $r'_{state}(s,i) = r_{state}(s)$ for all $s \in S$ and $0 \leqslant i \leqslant k{+}1$, and $r'_{action}((s,i),a) = r_{action}(s,a)$ for all $s \in S$, $1 \leqslant i \leqslant k{+}1$, and $a \in \alpha_{\mathcal{M}}$. In this case, we have that $\mathbb{E}^{\min}_{(s,0)}(F^{T'}_{r'})$ (respectively $\mathbb{E}^{\max}_{(s,0)}(F^{T'}_{r'})$) in $\mathcal{M}'$ equals $\mathbb{E}^{\min}_s(C^{\leqslant k}_r)$ (respectively $\mathbb{E}^{\max}_s(C^{\leqslant k}_r)$) in $\mathcal{M}$.

6 PCTL Model Checking

In this section, we discuss the use of temporal logic to express and verify more complex properties of systems than just reachability or reward-based properties. We then show how the techniques introduced in Sections 4 and 5 can be used to perform model checking of these properties.

6.1 The Logic PCTL

PCTL (Probabilistic Computational Tree Logic) [54,6] is a probabilistic extension of the temporal logic CTL [33]. PCTL is used to express properties of both DTMCs [54] and MDPs [6]. Here, we focus on MDPs. In the last part of this section, we will extend PCTL to reward-based properties and, in Section 7, we will discuss the alternative temporal logic LTL (linear temporal logic).

Definition 12 (PCTL syntax). *The syntax of PCTL is as follows:*

$$\begin{aligned}\phi &::= \mathtt{true} \;\big|\; c \;\big|\; \phi \wedge \phi \;\big|\; \neg\phi \;\big|\; \mathtt{P}_{\bowtie p}[\psi] \\ \psi &::= \mathtt{X}\,\phi \;\big|\; \phi\ \mathtt{U}^{\leqslant k}\ \phi \;\big|\; \phi\ \mathtt{U}\ \phi\end{aligned}$$

where c is an atomic proposition, $\bowtie \in \{\leqslant, <, \geqslant, >\}$, $p \in [0,1]$ and $k \in \mathbb{N}$.

PCTL formulas are interpreted over an MDP and we assume that the atomic propositions c are taken from the set AP used to label its states.

In the syntax above, we distinguish between state formulas ϕ and path formulas ψ, which are evaluated over states and paths, respectively. A property of a model will always be expressed as a state formula; path formulas only occur as the parameter of the *probabilistic path operator* $\mathtt{P}_{\bowtie p}[\psi]$. Intuitively, a state s satisfies $\mathtt{P}_{\bowtie p}[\psi]$ if, under any adversary, the probability of taking a path from s satisfying path formula ψ is in the interval specified by $\bowtie p$.

As path formulas, we allow the $\mathtt{X}$ (*next*), $\mathtt{U}^{\leqslant k}$ (*bounded until*) and $\mathtt{U}$ (*until*) operators, which are standard in temporal logics. Intuitively: $\mathtt{X}\,\phi$ is true if ϕ is satisfied in the next state; $\phi_1\ \mathtt{U}^{\leqslant k}\ \phi_2$ is true if ϕ_2 is satisfied within k time-steps and ϕ_1 holds up until that point; and $\phi_1\ \mathtt{U}\ \phi_2$ is true if ϕ_2 is satisfied at some point in the future and ϕ_1 holds up until then.

Semantics. To give the semantics of PCTL, we must first specify a class of adversaries Adv. More precisely, the satisfaction relation is parameterised by Adv and a PCTL formula is satisfied in a state s if it is satisfied under *all* adversaries $\sigma \in Adv$. In practice, Adv is usually taken to be the set $Adv_{\mathcal{M}}$ of all adversaries. The formal semantics of PCTL is as follows.

Definition 13 (PCTL semantics). *Let $\mathcal{M} = (S, \overline{s}, \alpha_{\mathcal{M}}, \delta_{\mathcal{M}}, L)$ be an MDP, Adv a class of adversaries of $\mathcal{M}$ and $s \in S$. The satisfaction relation $\models_{Adv}$ of PCTL is defined inductively by:*

$$\begin{array}{lll} s \models_{Adv} \mathtt{true} & & \textit{always} \\ s \models_{Adv} c & \iff & c \in L(s) \\ s \models_{Adv} \phi_1 \wedge \phi_2 & \iff & s \models_{Adv} \phi_1 \wedge s \models_{Adv} \phi_2 \\ s \models_{Adv} \neg\phi & \iff & s \not\models_{Adv} \phi \\ s \models_{Adv} \mathtt{P}_{\bowtie p}[\psi] & \iff & Pr^{\sigma}_{\mathcal{M},s}(\{\pi \in IPath_{\mathcal{M},s} \mid \pi \models_{Adv} \psi\}) \bowtie p \textit{ for all } \sigma \in Adv \end{array}$$

where, for any $\pi \in IPath_{\mathcal{M}}$:

$$\begin{array}{lcl} \pi \models_{Adv} \mathtt{X}\,\phi & \iff & \pi(1) \models_{Adv} \phi \\ \pi \models_{Adv} \phi_1\ \mathtt{U}^{\leqslant k}\ \phi_2 & \iff & \exists i \leqslant k\ .\ \big(\pi(i) \models_{Adv} \phi_2 \wedge \pi(j) \models_{Adv} \phi_1\ \forall j < i\big) \\ \pi \models_{Adv} \phi_1\ \mathtt{U}\ \phi_2 & \iff & \exists k \geqslant 0\ .\ \pi \models_{Adv} \phi_1\ \mathtt{U}^{\leqslant k}\ \phi_2\,. \end{array}$$

As for probabilistic reachability in Section 4, it is straightforward to show that the set of paths satisfying any PCTL path formula ψ is measurable [84,11].

With slight abuse of notation, we will use $Pr^{\sigma}_{\mathcal{M},Adv,s}(\psi)$ to denote the probability that a path from s satisfies path formula ψ under adversary σ:

$$Pr^{\sigma}_{\mathcal{M},Adv,s}(\psi) \stackrel{\text{def}}{=} Pr^{\sigma}_{\mathcal{M},s}(\{\pi \in IPath_{\mathcal{M},s} \mid \pi \models_{Adv} \psi\})$$

and define the minimum and maximum probabilities of satisfying the formula under the adversaries Adv for a starting state s:

$$\begin{array}{l} Pr^{\min}_{\mathcal{M},Adv,s}(\psi) \stackrel{\text{def}}{=} \inf_{\sigma \in Adv} Pr^{\sigma}_{\mathcal{M},Adv,s}(\psi) \\ Pr^{\max}_{\mathcal{M},Adv,s}(\psi) \stackrel{\text{def}}{=} \sup_{\sigma \in Adv} Pr^{\sigma}_{\mathcal{M},Adv,s}(\psi). \end{array}$$

Where clear from the context, we will omit the subscripts $\mathcal{M}$ and/or Adv.

Additional Operators. From the basic PCTL syntax, we can derive several other useful operators. Among these are the well known logical equivalences:

$$\begin{array}{rcl} \mathtt{false} & \equiv & \neg\mathtt{true} \\ \phi_1 \vee \phi_2 & \equiv & \neg(\neg\phi_1 \wedge \neg\phi_2) \\ \phi_1 \rightarrow \phi_2 & \equiv & \neg\phi_1 \vee \phi_2 \end{array}$$

We also allow path formulas to contain the $\mathtt{F}$ (*future*) operator (often written as $\Diamond$), which is common in temporal logics. Intuitively, $\mathtt{F}\ \phi$ means that ϕ is eventually satisfied, and its bounded variant $\mathtt{F}^{\leqslant k}\ \phi$ means that ϕ is satisfied within k steps. These can be expressed in terms of the PCTL until and bounded until operators as follows:

$$\mathtt{F}\ \phi \equiv \mathtt{true}\ \mathtt{U}\ \phi \quad \text{and} \quad \mathtt{F}^{\leqslant k}\ \phi \equiv \mathtt{true}\ \mathtt{U}^{\leqslant k}\ \phi\,.$$

Similarly, we add a $\mathtt{G}$ (*globally*) operator (often written as $\Box$), where $\mathtt{G}\ \phi$ intuitively means that ϕ is always satisfied. It too has a bounded variant $\mathtt{G}^{\leqslant k}\ \phi$, which means that ϕ is continuously satisfied for k steps. These operators can be expressed using the equivalences:

$$\mathtt{G}\ \phi \equiv \neg(\mathtt{F}\ \neg\phi) \quad \text{and} \quad \mathtt{G}^{\leqslant k}\ \phi \equiv \neg(\mathtt{F}^{\leqslant k}\ \neg\phi)\,.$$

Strictly speaking, the $\mathtt{G}$ and $\mathtt{G}^{\leqslant k}$ operators cannot be derived from the basic syntax of PCTL since we do not allow negation in path formulas. However, it can be shown that, for example:

$$\mathtt{P}_{\geqslant p}[\mathtt{G}\ \phi] \equiv \mathtt{P}_{\leqslant 1-p}[\mathtt{F}\ \neg\phi]$$

See Section 7.2 (page 87) for an explanation of the above equivalence.

Examples. Some examples of PCTL formulas, taken from case studies, are:

- $\mathtt{P}_{<0.05}[\mathtt{F}\ (\mathit{sensor\_fail}_1 \wedge \mathit{sensor\_fail}_2)]$ – "the probability of simultaneous failures occurring in both sensors is less than 0.05";
- $\mathtt{P}_{\geqslant 0.8}[\mathtt{F}^{\leqslant k}\ \mathit{ack}_n]$ – "the probability that the sender has received n acknowledgements within k clock-ticks is at least 0.8";
- $\mathtt{P}_{<0.4}[\neg \mathit{fail}_A\ \mathtt{U}\ \mathit{fail}_B]$ – "the probability that component B fails before component A is less than 0.4";
- $\neg \mathit{oper} \rightarrow \mathtt{P}_{\geqslant 1}[\mathtt{F}\ (\mathtt{P}_{>0.99}[\mathtt{G}^{\leqslant 100}\ \mathit{oper}])]$ – "if the system is not operational, it almost surely reaches a state from which it has a greater than 0.99 chance of staying operational for 100 time units".

Extensions of PCTL. A commonly used extension of PCTL, which derives from the PRISM model checker [56], is the addition of *quantitative* versions of the P operator. Rather than stating that the probability of some path formula ψ being satisfied is always above or below a threshold, *quantitative* properties simply ask: "what is the minimum/maximum probability of ψ holding?". For this, we add the operators $\mathtt{P}_{\mathtt{min}=?}[\psi]$ and $\mathtt{P}_{\mathtt{max}=?}[\psi]$ and, adapting the examples from above, we can express:

- $\mathtt{P}_{\mathtt{min}=?}[\mathtt{F}^{\leqslant k}\ \mathit{ack}_n]$ - "what is the minimum probability that the sender has received n acknowledgements within k clock-ticks?";
- $\mathtt{P}_{\mathtt{max}=?}[\neg \mathit{fail}_A\ \mathtt{U}\ \mathit{fail}_B]$ - "what is the maximum probability that component B fails before component A?".

Of course, these operators cannot be nested within PCTL formulas, like in the fourth example from the earlier list. Thus, in the two examples above, $\mathtt{P}_{\mathtt{min}=?}[\psi]$ or $\mathtt{P}_{\mathtt{max}=?}[\psi]$ is the outermost operator of the formula. As will be seen in the next section, the process of model checking a PCTL formula $\mathtt{P}_{\bowtie p}[\psi]$ requires computation of the probabilities $Pr_s^{\min}(\psi)$ or $Pr_s^{\max}(\psi)$ anyway, so these quantitative properties are no more expensive to analyse.

In addition, when writing specifications for MDPs in PCTL, it may sometimes be useful to consider the *existence* of an adversary that satisfies a particular property, rather than stating that *all* adversaries satisfy it. For simple formulas, this can be done via translation to a dual property. For example, verifying that "there exists an adversary σ for which, from state s, the probability of satisfying ψ is at least p" is equivalent to model checking the PCTL formula $\neg\mathtt{P}_{<p}[\psi]$, which states "it is not the case that under all adversaries the probability of satisfying ψ from state s is less than p". Later, in Section 8, we will discuss the feasibility of checking the existence of adversaries satisfying more complex formulas.

6.2 PCTL Model Checking

Model checking a PCTL formula ϕ on an MDP $\mathcal{M}$ amounts to checking which states of $\mathcal{M}$ satisfy ϕ. The basic structure of the algorithm for PCTL model checking [54,18] is similar to the model checking algorithm for the temporal logic CTL [33]. First, we construct a parse tree of the formula ϕ. Each node of

$$\begin{tikzcd}
 & & \mathtt{P}_{<0.5}[\mathtt{X}\neg\mathtt{P}_{\geqslant 1}[a\ \mathtt{U}\ (b \wedge c)]] \arrow[d] & & \\
 & & \neg\mathtt{P}_{\geqslant 1}[a\ \mathtt{U}\ (b \wedge c)] \arrow[d] & & \\
 & & \mathtt{P}_{\geqslant 1}[a\ \mathtt{U}\ (b \wedge c)] \arrow[dl] \arrow[dr] & & \\
 & a & & b \wedge c \arrow[dl] \arrow[dr] & \\
 & & b & & c
\end{tikzcd}$$

Fig. 7. The parse tree for a formula $\mathtt{P}_{<0.5}[\mathtt{X}\,\neg\mathtt{P}_{\geqslant 1}[a\ \mathtt{U}\ (b \wedge c)]]$

the tree is labelled with a subformula of ϕ, the root is labelled with ϕ itself and the leaves are labelled with either `true` or atomic propositions (an example parse tree is depicted in Figure 7). Working upwards towards the root, we recursively compute the set of states satisfying each subformula, and at the end we have determined the set of states satisfying ϕ. For a given class of adversaries Adv, let $Sat_{Adv}(\phi)$ denote the set $\{s \in S \mid s \models_{Adv} \phi\}$ of all states satisfying the formula ϕ under the class of adversaries Adv. Letting $\rhd \in \{\geqslant, >\}$ and $\lhd \in \{\leqslant, <\}$, the algorithm for PCTL formulas can be summarised as follows:

$$\begin{aligned}
Sat_{Adv}(\mathtt{true}) &= S \\
Sat_{Adv}(c) &= \{s \,|\, c \in L(s)\} \\
Sat_{Adv}(\neg\,\phi) &= S \backslash Sat_{Adv}(\Phi) \\
Sat_{Adv}(\phi_1 \wedge \phi_2) &= Sat_{Adv}(\phi_1) \cap Sat_{Adv}(\phi_2) \\
Sat_{Adv}(\mathtt{P}_{\rhd p}[\psi]) &= \{s \in S \,|\, Pr^{\min}_{Adv,s}(\psi) \rhd p\} \\
Sat_{Adv}(\mathtt{P}_{\lhd p}[\psi]) &= \{s \in S \,|\, Pr^{\max}_{Adv,s}(\psi) \lhd p\}\,.
\end{aligned}$$

Obviously, the most difficult part in the above algorithm is the computation of the probability bounds $Pr^{\min}_{Adv,s}(\psi)$ and $Pr^{\max}_{Adv,s}(\psi)$. In what follows, we describe how to compute these probability bounds for the different possible path formulas ψ when Adv is the set $Adv_{\mathcal{M}}$ of *all* adversaries of the MDP, and therefore omit Adv from the subscript. An alternative would be to consider the class of *fair* adversaries. We do not discuss the issue of *fairness* when model checking MDPs in this tutorial; for details, see e.g. [2,8,12].

The "Next" Operator. If $\psi = \mathtt{X}\,\phi$, then it follows that:

$$\begin{aligned}
Pr^{\min}_{s}(\mathtt{X}\,\phi) &= \min\nolimits_{a \in A(s)} \textstyle\sum_{s' \in Sat(\phi)} \delta_{\mathcal{M}}(s,a)(s') \\
Pr^{\max}_{s}(\mathtt{X}\,\phi) &= \max\nolimits_{a \in A(s)} \textstyle\sum_{s' \in Sat(\phi)} \delta_{\mathcal{M}}(s,a)(s')
\end{aligned}$$

both of which can be computed easily. An optimal memoryless adversary σ can be constructed by putting $\sigma(s) = [a \mapsto 1]$ where a is an action that minimises (or maximises) $\sum_{s' \in Sat(\phi)} \delta_{\mathcal{M}}(s,a)(s')$.

The "Bounded Until" Operator. Consider the case $\psi = \phi_1 \ \mathtt{U}^{\leqslant k} \ \phi_2$ for minimum probabilities, i.e. computation of $Pr_s^{\min}(\phi_1 \ \mathtt{U}^{\leqslant k} \ \phi_2)$ for all states s. These can be computed by solving the following equations:

$$x_s^\ell = \begin{cases} 1 & \text{if } s \in Sat(\phi_2) \\ 0 & \text{if } s \notin (Sat(\phi_1) \cup Sat(\phi_2)) \\ 0 & \text{if } s \in (Sat(\phi_1) \backslash Sat(\phi_2)) \text{ and } \ell{=}0 \\ \min_{a \in A(s)} \sum_{s' \in S} \delta_{\mathcal{M}}(s,a)(s') \cdot x_{s'}^{\ell-1} & \text{otherwise} \end{cases}$$

and setting $Pr_s^{\min}(\phi_1 \ \mathtt{U}^{\leqslant k} \ \phi_2) = x_s^k$. Like for reward-based properties, an optimal adversary can be constructed on-the-fly by remembering the action:

$$a_s^\ell = \arg\min_{a \in A(s)} \textstyle\sum_{s' \in S} \delta_{\mathcal{M}}(s,a)(s') \cdot x_{s'}^{\ell-1}$$

for all $0{\leqslant}\ell{\leqslant}k$ and $s \in S$, and setting $\sigma(\rho) = [a_\rho {\mapsto} 1]$ where $a_\rho = a_{last(\rho)}^{k-|\rho|}$ for all $\rho \in FPath$ such that $|\rho|{\leqslant}k{-}1$. For maximum probabilities, we simply replace "min" with "max" in the equations above.

The "Unbounded Until" Operator. It remains to give a procedure that computes the minimum and maximum probabilities when $\psi = \phi_1 \ \mathtt{U} \ \phi_2$. The approach is based on the probabilistic reachability computation given in Section 4 for the target $T{=}Sat(\phi_2)$. However, here, we also need to ensure that ϕ_1 holds before satisfying ϕ_2. Thus, we restrict attention to paths that remain in the set of states $Sat(\phi_1)$ before reaching the target. Formally, this can be done by preprocessing the MDP in the following way. For each state $s \in S \backslash Sat(\phi_1)$ and action $a \in A(s)$ we change $\delta_{\mathcal{M}}(s,a)$ to $[s{\mapsto}1]$. It then follows that $Pr_s^{\min}(\phi_1 \ \mathtt{U} \ \phi_2)$ (respectively, $Pr_s^{\max}(\phi_1 \ \mathtt{U} \ \phi_2)$) in the original MDP equals $Pr_s^{\min}(reach(Sat(\phi_2)))$ (respectively, $Pr_s^{\max}(reach(Sat(\phi_2)))$) in the preprocessed MDP. The solution methods such as value iteration or policy iteration presented in Section 4 can therefore be applied, as well as the adversary generation.

Example 11. Consider again the example MDP from Figure 2, together with the formula $\mathtt{P}_{<1}[\mathtt{X}\,(\mathtt{P}_{\geqslant 0.5}[\neg fail \ \mathtt{U} \ init])]$. We start with the analysis of the innermost subformulas and obtain $Sat(fail){=}\{s_3\}$ and $Sat(init){=}\{s_0\}$. Then, we proceed with $\neg fail$ and find $Sat(\neg fail){=}\{s_0, s_1, s_2\}$. Next, considering the formula $\mathtt{P}_{\geqslant 0.5}[\neg fail \ \mathtt{U} \ init]$, we modify the MDP so that in s_3 we loop on action *reset*, and then compute the minimum probability of reaching $\{s_0\}$. Computing $S^0_{\min}$ and $S^1_{\min}$ yields $S^0_{\min}{=}\{s_1, s_2, s_3\}$ and $S^1_{\min}{=}\{s_0\}$, which gives values for all states, and hence $Sat(\mathtt{P}_{\geqslant 0.5}[\neg fail \ \mathtt{U} \ init]) = \{s_0\}$. Finally, for $\mathtt{P}_{<1}[\mathtt{X}\,(\mathtt{P}_{\geqslant 0.5}[\neg fail \ \mathtt{U} \ init])]$, we compute $Pr_s^{\max}(\mathtt{X}\,(\mathtt{P}_{\geqslant 0.5}[\neg fail \ \mathtt{U} \ init]))$ for $s \in \{s_0, s_1, s_2, s_3\}$, and obtain the values $(0, \max\{0.7{\cdot}1 + 0.3{\cdot}0, 0.5{\cdot}0 + 0.5{\cdot}0\}, 0, \max\{0, 1\}) = (0, 0.7, 0, 1)$, yielding $Sat(\mathtt{P}_{<1}[\mathtt{X}\,(\mathtt{P}_{\geqslant 0.5}[\neg fail \ \mathtt{U} \ init])]) = \{s_0, s_1, s_2\}$. ■

6.3 Extending PCTL with Rewards

We can extend the definition of PCTL to include the reward-related properties introduced in Section 5. Here, we present the extension of PCTL used in PRISM.

More expressive logics for reward-based properties of MDPs can be found in [1]. The syntax for state formulas of PCTL becomes:

$$\phi ::= \texttt{true} \mid c \mid \phi \wedge \phi \mid \neg\phi \mid \texttt{P}_{\bowtie p}[\psi] \mid \texttt{R}^{r}_{\bowtie x}[\texttt{I}^{=k}] \mid \texttt{R}^{r}_{\bowtie x}[\texttt{C}^{\leqslant k}] \mid \texttt{R}^{r}_{\bowtie x}[\texttt{F}\ \phi]$$

where c is an atomic proposition, $\bowtie \in \{\leqslant, <, \geqslant, >\}$, $p \in [0,1]$, r is a reward structure, $x \in \mathbb{R}_{\geqslant 0}$ and $k \in \mathbb{N}$. The semantics of the previously introduced operators remains unchanged (see Definition 13), while the semantics of the new operators is defined as follows:

$$
\begin{aligned}
s \models_{Adv} \texttt{R}^{r}_{\bowtie x}[\texttt{I}^{=k}] &\iff \mathbb{E}^{\sigma}_{\mathcal{M},s}(I^{=k}_{r_{state}}) \bowtie x \text{ for all } \sigma \in Adv\text{, where } r{=}(r_{state}, r_{action}) \\
s \models_{Adv} \texttt{R}^{r}_{\bowtie x}[\texttt{C}^{\leqslant k}] &\iff \mathbb{E}^{\sigma}_{\mathcal{M},s}(C^{\leqslant k}_{r}) \bowtie x \text{ for all } \sigma \in Adv \\
s \models_{Adv} \texttt{R}^{r}_{\bowtie x}[\texttt{F}\ \phi] &\iff \mathbb{E}^{\sigma}_{\mathcal{M},s}(F^{Sat_{Adv}(\phi)}_{r}) \bowtie x \text{ for all } \sigma \in Adv\,.
\end{aligned}
$$

We can reuse the basic model checking algorithm from above, but we need to extend it to deal with the new operators. Letting $\rhd \in \{\geqslant, >\}$ and $\lhd \in \{\leqslant, <\}$, we have to compute the following sets:

$$
\begin{aligned}
Sat_{Adv}(\texttt{R}^{r}_{\rhd x}[\texttt{I}^{=k}]) &= \{s \in S \mid \inf_{\sigma \in Adv} \mathbb{E}^{\sigma}_{s}(I^{=k}_{r_{state}}) \rhd x\} \\
Sat_{Adv}(\texttt{R}^{r}_{\lhd x}[\texttt{I}^{=k}]) &= \{s \in S \mid \sup_{\sigma \in Adv} \mathbb{E}^{\sigma}_{s}(I^{=k}_{r_{state}}) \lhd x\} \\
Sat_{Adv}(\texttt{R}^{r}_{\rhd x}[\texttt{C}^{\leqslant k}]) &= \{s \in S \mid \inf_{\sigma \in Adv} \mathbb{E}^{\sigma}_{s}(C^{\leqslant k}_{r}) \rhd x\} \\
Sat_{Adv}(\texttt{R}^{r}_{\lhd x}[\texttt{C}^{\leqslant k}]) &= \{s \in S \mid \sup_{\sigma \in Adv} \mathbb{E}^{\sigma}_{s}(C^{\leqslant k}_{r}) \lhd x\} \\
Sat_{Adv}(\texttt{R}^{r}_{\rhd x}[\texttt{F}\ \phi]) &= \{s \in S \mid \inf_{\sigma \in Adv} \mathbb{E}^{\sigma}_{s}(F^{Sat_{Adv}(\phi)}_{r}) \rhd x\} \\
Sat_{Adv}(\texttt{R}^{r}_{\lhd x}[\texttt{F}\ \phi]) &= \{s \in S \mid \sup_{\sigma \in Adv} \mathbb{E}^{\sigma}_{s}(F^{Sat_{Adv}(\phi)}_{r}) \lhd x\}\,.
\end{aligned}
$$

Hence, if Adv is the set of all adversaries of $\mathcal{M}$, then we need to compute the minimum and maximum expected values of the random variables $I^{=k}_{r_{state}}$, $C^{\leqslant k}_{r}$ and $F^{Sat_{Adv}(\phi)}_{r}$, which can be achieved using the algorithms of Section 5. Like for the $\texttt{P}$ operator, we can also consider *quantitative* versions $\texttt{R}^{r}_{\texttt{min=?}}[\cdot]$ and $\texttt{R}^{r}_{\texttt{max=?}}[\cdot]$ of $\texttt{R}$ which ask: "what is the minimum/maximum expected reward?".

6.4 Complexity

The model checking algorithms for PCTL on MDPs [35,18] are polynomial in the size of the model and linear in the size of the formula, where the sizes of the parameters are defined as follows. The size of an MDP $\mathcal{M}$, denoted $|\mathcal{M}|$, equals the total number of nondeterministic choices (since we require $A(s)$ to be non-empty for all states s, it is always the case that this is greater than the number of states). The size of a formula ϕ, denoted $|\phi|$, equals the number of logical connectives and temporal operators in the formula plus the sum of the sizes of the temporal operators.

Due to the recursive nature of the model checking algorithm, we perform model checking for each of the $|\phi|$ operators of ϕ individually. The most expensive cases concern computation of minimum and maximum reachability probabilities or expected cumulative reward to reach a target, for which we must solve a linear optimisation problem of size $|\mathcal{M}|$. Using, for example, the ellipsoid method, this can be performed in polynomial time.

Generally, to simplify the complexity analysis, we will ignore issues regarding number representations, e.g. of probabilities in the MDP or constants in a PCTL formula. If, for example, we took the size of the formula to also include the binary encoding of the numbers k in the bounded until operators, the complexity of model checking algorithm would be exponential in the size of the formula.

7 Linear-Time Probabilistic Model Checking

The logic PCTL described in the previous section is a *branching-time* logic. Notice that, when referring to the probability of an event occurring (using the P operator), only a single temporal operator (such as U, F or G) can be used. The only way to combine temporal operators is to use a nested formula, such as $\mathtt{P}_{<0.2}[\mathtt{F}\ \mathtt{P}_{\geqslant 0.9}[\mathtt{G}\ a]]$. However, the meaning of such formulas can be subtle as each appearance of the P operator has a separate quantification over adversaries.

In this section, we consider *linear-time* properties for MDPs, in which the probability of more complex events can be expressed. We will discuss two distinct classes: *probabilistic safety properties* and properties expressed in *linear temporal logic* (LTL), which are in fact a special case of *ω-regular properties.* In each case, computing the required probabilities can be acheived through the use of *automata*: either finite automata, for probabilistic safety properties, or ω-automata such as Rabin automata, for LTL and ω-regular properties.

Another important point to make is that, in this section, we will consider linear-time properties expressed in terms of the *actions* that label the transitions of an MDP, rather than the *atomic propositions* labelling its states, as was done for PCTL in the previous section. In fact, either approach can be taken and the model checking process is very similar. In this presentation, we opt for action-based properties since these are required for the compositional probabilistic model checking techniques that we discuss in Section 9. See, for example, [35,11,32] for details of the state-based approach.

7.1 Probabilistic Safety Properties

To define probabilistic safety properties, we first recall the definitions of *deterministic finite automata* and *regular safety properties.*

Definition 14 (Deterministic finite automaton). *A* deterministic finite automaton *(DFA) is a tuple $\mathcal{A} = (Q, \overline{q}, \alpha_{\mathcal{A}}, \delta_{\mathcal{A}}, F)$, comprising a finite set of states Q, initial state $\overline{q} \in Q$, finite alphabet $\alpha_{\mathcal{A}}$, transition function $\delta_{\mathcal{A}} : Q \times \alpha_{\mathcal{A}} \to Q$ and accepting states $F \subseteq Q$.*

We say that $\mathcal{A}$ is *complete* (or *total*) if the transition function $\delta_{\mathcal{A}}$ is total. A DFA $\mathcal{A}$ defines the regular language $\mathcal{L}(\mathcal{A}) \subseteq (\alpha_{\mathcal{A}})^*$ where $a_0 \ldots a_k \in \mathcal{L}(\mathcal{A})$ if and only if $\overline{q} \xrightarrow{a_0} q_1 \xrightarrow{a_1} \cdots \xrightarrow{a_k} q_{k+1}$ is a path of $\mathcal{A}$ (i.e. $\delta_{\mathcal{A}}(\overline{q}, a_0){=}q_1$ and $\delta_{\mathcal{A}}(q_i, a_i){=}q_{i+1}$ for all $1{\leqslant}i{\leqslant}k$) and $q_{k+1} \in F$.

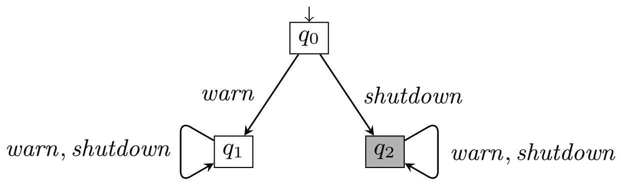


Fig. 8. DFA $\mathcal{A}_A^{err}$ for the regular safety property Φ_A in Example 12

Definition 15 (Regular safety property). *A* regular safety property Φ_P *represents a set of infinite words* $\mathcal{L}(\Phi_P) \subseteq (\alpha_P)^\omega$ *over an alphabet* α_P*, which is characterised by a regular language of "bad prefixes", i.e. finite words of which any extension is* not *in* $\mathcal{L}(\Phi_P)$*. We represent* Φ_P *by an* error automaton $\mathcal{A}_P^{err}$*: a (complete) DFA over* α_P *that stores these bad prefixes. Formally:*

$$\mathcal{L}(\Phi_P) \stackrel{\text{def}}{=} \{w \in (\alpha_P)^\omega \mid \textit{no prefix of } w \textit{ is in } \mathcal{L}(\mathcal{A}_P^{err})\} .$$

Regular safety properties can capture properties such as:

– "event A always occurs before event B";
– "a system failure never occurs";
– "termination occurs within at most k steps".

Example 12. Consider the regular safety property Φ_A with the meaning "*warn* occurs before *shutdown*" for a model whose alphabet includes *warn* and *shutdown*. Figure 8 shows the corresponding DFA $\mathcal{A}_A^{err}$ where accepting states are shaded. The "bad prefixes" are the finite words that do not begin with *warn*. ■

To determine the probability that a safety property Φ_P is satisfied, we look at the set of paths whose traces, when restricted to α_P, are in $\mathcal{L}(\Phi_P)$. Consider an MDP $\mathcal{M}$ and regular safety property Φ_P with $\alpha_P \subseteq \alpha_\mathcal{M}$. For any adversary σ and state s of $\mathcal{M}$, we define the probability, under σ, of Φ_P being satisfied when starting from s as:

$$Pr^\sigma_{\mathcal{M},s}(\Phi_P) \stackrel{\text{def}}{=} Pr^\sigma_{\mathcal{M},s}(\{\pi \in IPath_{\mathcal{M},s} \mid tr(\pi)\restriction_{\alpha_P} \in \mathcal{L}(\Phi_P) \cup \mathcal{L}^*(\Phi_P)\})$$

where $w\restriction_\alpha$ is the projection of word w onto the alphabet α and $\mathcal{L}^*(\Phi_P) = \{w \in (\alpha_P)^* \mid \text{no prefix of } w \text{ is in } \mathcal{L}(\mathcal{A}_P^{err})\}$. The inclusion of $\mathcal{L}^*(\Phi_P)$ is required because the projection can return finite words. As stated earlier, the set of paths that satisfy Φ_P is always measurable, so the notation is well-defined. As for other classes of property, we also define:

$$Pr^{\min}_{\mathcal{M},s}(\Phi_P) \stackrel{\text{def}}{=} \inf_{\sigma \in Adv_\mathcal{M}} Pr^\sigma_{\mathcal{M},s}(\Phi_P)$$
$$Pr^{\max}_{\mathcal{M},s}(\Phi_P) \stackrel{\text{def}}{=} \sup_{\sigma \in Adv_\mathcal{M}} Pr^\sigma_{\mathcal{M},s}(\Phi_P) .$$

We can now introduce, using the probability bound operator $\mathtt{P}_{\geqslant p}[\cdot]$, the class of *probabilistic safety properties.*

Definition 16 (Probabilistic safety property). *A probabilistic safety property* $\mathtt{P}_{\geqslant p}[\Phi_P]$ *comprises a regular safety property* Φ_P *and a (lower) probability bound* $p \in (0,1]$*. A state* s *of an MDP* $\mathcal{M}$ *satisfies the property, denoted* $s \models \mathtt{P}_{\geqslant p}[\Phi_P]$*, if the probability of satisfying* Φ_P *is at least* p *for all adversaries:*

$$\begin{aligned} s \models \mathtt{P}_{\geqslant p}[\Phi_P] &\iff \forall \sigma \in Adv_{\mathcal{M}} \ . \ Pr^{\sigma}_{\mathcal{M},s}(\Phi_P) \geqslant p \\ &\iff Pr^{\min}_{\mathcal{M},s}(\Phi_P) \geqslant p \,. \end{aligned}$$

Probabilistic safety properties can be used to capture a variety of useful properties of MDPs; for example:

– "event A always occurs before event B with probability at least 0.9";
– "the probability of a system failure occurring is at most 0.02";
– "the probability of terminating within k time-units is at least 0.75".

Notice that, in the second example above, we express the property in terms of the *maximum* probability of the safety property *not* holding, rather than the (equivalent) *minimum* probability that it *does* hold. This equivalence is also used when model checking a probabilistic safety property $\mathtt{P}_{\geqslant p}[\Phi_P]$. We reduce the problem of computing the probability $Pr^{\min}_{\mathcal{M},s}(\Phi_P)$ for each state s of an MDP to the problem of computing *maximum* reachability probabilities in the *product* $\mathcal{M}\otimes\mathcal{A}^{err}_P$ of the MDP $\mathcal{M}$ and an error automaton $\mathcal{A}^{err}_P$ for Φ_P.

Definition 17 (MDP-DFA product). *If* $\mathcal{M}=(S,\overline{s},\alpha_{\mathcal{M}},\delta_{\mathcal{M}},L_{\mathcal{M}})$ *is an MDP,* Φ_P *is a regular safety property with* $\alpha_P \subseteq \alpha_{\mathcal{M}}$ *and* $\mathcal{A}^{err}_P=(Q,\overline{q},\alpha_P,\delta_P,F)$ *is an error automaton for* Φ_P*, then the* product MDP*, denoted* $\mathcal{M}\otimes\mathcal{A}^{err}_P$*, is given by* $(S\times Q,(\overline{s},\overline{q}),\alpha_{\mathcal{M}},\delta_{\mathcal{M}\otimes\mathcal{A}^{err}_P},L_{\mathcal{M}\otimes\mathcal{A}^{err}_P})$ *where, for each* $(s,q) \in S\times Q$ *and* $a \in \alpha_{\mathcal{M}}$*:*

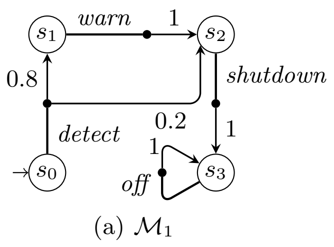
– $\delta_{\mathcal{M}\otimes\mathcal{A}^{err}_P}((s,q),a) = \begin{cases} \delta_{\mathcal{M}}(s,a)\times[\delta_P(q,a)\mapsto 1] & \text{if } a \in A(s)\cap\alpha_P \\ \delta_{\mathcal{M}}(s,a)\times[q\mapsto 1] & \text{if } a \in A(s)\backslash\alpha_P \\ \text{undefined} & \text{otherwise} \end{cases}$

– $L_{\mathcal{M}\otimes\mathcal{A}^{err}_P}(s,q) = \begin{cases} L_{\mathcal{M}}(s)\cup\{err_P\} & \text{if } q \in F \\ L_{\mathcal{M}}(s) & \text{otherwise.} \end{cases}$

Intuitively, the product records both the state of the MDP $\mathcal{M}$ and the state of the DFA $\mathcal{A}^{err}_P$, based on the actions seen so far in the history of $\mathcal{M}$. The labelling is also modified by marking states corresponding to accepting states of $\mathcal{A}^{err}_P$ with the new atomic proposition err_P (we will use this later in Section 9). Crucially, because the automaton is deterministic, each path through $\mathcal{M}\otimes\mathcal{A}^{err}_P$ corresponds to a unique path in each of $\mathcal{M}$ and $\mathcal{A}^{err}_P$. Consequently: (i) the probability of events in $\mathcal{M}$ is preserved in $\mathcal{M}\otimes\mathcal{A}^{err}_P$; and (ii) each path of $\mathcal{M}\otimes\mathcal{A}^{err}_P$ that corresponds to a path of $\mathcal{M}$ that violates Φ_P contains a state in $S\times F$.

Proposition 1 ([69]). *If* $\mathcal{M}=(S,\overline{s},\alpha_{\mathcal{M}},\delta_{\mathcal{M}},L_{\mathcal{M}})$ *is an MDP,* $s \in S$*,* Φ_P *is a safety property such that* $\alpha_P \subseteq \alpha_{\mathcal{M}}$*, and* $\mathcal{A}^{err}_P$ *is an error automaton for* Φ_P *with accepting states* F*, then:*

$$Pr^{\min}_{\mathcal{M},s}(\Phi_P) = 1 - Pr^{\max}_{\mathcal{M}\otimes\mathcal{A}^{err}_P,(s,\overline{q})}(reach(S\times F))\,.$$



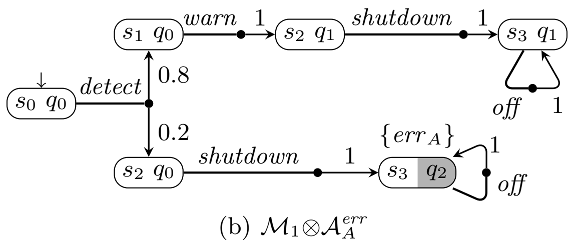


Fig. 9. An MDP $\mathcal{M}_1$ and the product $\mathcal{M}_1 \otimes \mathcal{A}_A^{err}$ with the DFA $\mathcal{A}_A^{err}$ from Figure 8

Thus, checking satisfaction of $\mathtt{P}_{\geqslant p}[\Phi_P]$ on MDP $\mathcal{M}$ can be achieved by applying the techniques of Section 4 to the product MDP $\mathcal{M} \otimes \mathcal{A}_P^{err}$.

Example 13. Consider the MDP $\mathcal{M}_1$ in Figure 9(a) and probabilistic safety property $\mathtt{P}_{\geqslant 0.8}[\Phi_A]$ where Φ_A is the regular safety property "*warn* occurs before *shutdown*" from Example 12, represented by the DFA $\mathcal{A}_A^{err}$ of Figure 8. The product $\mathcal{M}_1 \otimes \mathcal{A}_A^{err}$ is shown in Figure 9(b) and, using the techniques of Section 4, we find that $Pr^{\max}_{\mathcal{M}_1 \otimes \mathcal{A}_A^{err},(s_0,q_0)}(reach(S \times \{q_2\}))=0.2$. Therefore, using Proposition 1, we have $Pr^{\min}_{\mathcal{M}_1,s_0}(\Phi_A)=1-0.2=0.8$, yielding $s_0 \models \mathtt{P}_{\geqslant 0.8}[\Phi_A]$. ■

7.2 LTL and ω-Regular Properties

LTL (linear temporal logic) [75] is a widely used temporal logic that is particularly well suited for expressing long-run properties of systems. As discussed above, in this presentation, we use LTL to define properties of MDPs in terms of their action labels. For the remainder of this section, we assume an MDP $\mathcal{M}$ with the alphabet of action labels $\alpha_{\mathcal{M}}$.

Definition 18 (LTL syntax). *The syntax of LTL is defined by the grammar:*

$$\psi ::= \mathtt{true} \mid a \mid \psi \wedge \psi \mid \neg\psi \mid \mathtt{X}\,\psi \mid \psi\ \mathtt{U}\ \psi$$

where $a \in \alpha_{\mathcal{M}}$.

As for PCTL (see Section 6), we can derive additional operators $\vee$, $\rightarrow$, $\mathtt{F}$ and $\mathtt{G}$.

The satisfaction of an LTL formula ψ is given in terms of infinite words over the alphabet $\alpha_{\mathcal{M}}$. To give the semantics, we require some additional notation regarding words. For any infinite word $w = a_0a_1a_2\ldots$, let $w[i]$ denote the $(i+1)$th element a_i of w and let $w[i\ldots]$ denote the suffix $a_ia_{i+1}\ldots$ of w.

Definition 19 (LTL semantics). *Let w be an infinite word over the alphabet $\alpha_{\mathcal{M}}$. The satisfaction relation $\models$ for LTL is defined inductively by:*

$$\begin{array}{lll}
w \models \mathtt{true} & & \textit{always} \\
w \models a & \iff & w[0] = a \\
w \models \psi_1 \wedge \psi_2 & \iff & w \models \psi_1 \wedge w \models \psi_2 \\
w \models \neg\psi & \iff & w \not\models \psi \\
w \models \mathtt{X}\,\psi & \iff & w[1\ldots] \models \psi \\
w \models \psi_1\ \mathtt{U}\ \psi_2 & \iff & \exists i \in \mathbb{N}\ .\ \big(w[i\ldots] \models \psi_2 \wedge (\forall j<i\ .\ w[j\ldots] \models \psi_1)\big)\,.
\end{array}$$

Using this definition, the satisfaction of an LTL formula ψ by an infinite path π of the MDP $\mathcal{M}$ can be defined in terms of the trace of the path as follows:

$$\pi \models \psi \iff tr(\pi) \models \psi.$$

Example 14. Let us consider the following paths from the MDP in Figure 2:

$$\begin{aligned}\pi_1 &= s_0 \left(\xrightarrow{go} s_1 \xrightarrow{risk} s_3 \xrightarrow{reset} s_0 \right)^{\omega} \\ \pi_2 &= s_0 \xrightarrow{go} s_1 \xrightarrow{risk} s_3 \xrightarrow{reset} s_0 \xrightarrow{go} s_1 \xrightarrow{safe} s_2 \left(\xrightarrow{finish} s_2 \right)^{\omega}\end{aligned}$$

The formula $\mathtt{F}\ (reset \wedge (\mathtt{X}\,\mathtt{F}\ reset))$, which intuitively means that *reset* occurs at least twice in a path, is true in π_1, but not true in π_2. On the other hand, the formula $\mathtt{F}\ \mathtt{G}\ finish$, which says that, from some point on, we will only take the action *finish*, is true in π_2, but not in π_1. Other examples are the formula $\mathtt{X}\,\mathtt{X}\,(\neg risk\ \mathtt{U}\ finish)$, which is satisfied in π_2, but not in π_1, and the formula $\mathtt{G}\ (go \rightarrow \mathtt{X}\, risk)$, which is satisfied in π_1 but not in π_2. ■

It is now straightforward to define the probability of satisfying an LTL formula ψ when starting in a state s under an adversary σ of the MDP $\mathcal{M}$:

$$Pr^{\sigma}_{\mathcal{M},s}(\psi) \stackrel{\text{def}}{=} Pr^{\sigma}_{\mathcal{M},s}(\{\pi \in IPath_{\mathcal{M},s} \mid \pi \models \psi\})\,.$$

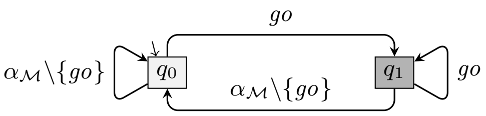
The set of paths satisfying an LTL formula ψ is always measurable [84,11]. As usual, we define the minimum and maximum probability of satisfaction over all adversaries of the MDP:

$$\begin{aligned} Pr^{\min}_{\mathcal{M},s}(\psi) &\stackrel{\text{def}}{=} \inf_{\sigma \in Adv_{\mathcal{M}}} Pr^{\sigma}_{\mathcal{M},s}(\psi) \\ Pr^{\max}_{\mathcal{M},s}(\psi) &\stackrel{\text{def}}{=} \sup_{\sigma \in Adv_{\mathcal{M}}} Pr^{\sigma}_{\mathcal{M},s}(\psi). \end{aligned}$$

Definition 20 (Probabilistic LTL specification). *A* probabilistic LTL specification *is a formula* $\mathtt{P}_{\bowtie p}[\psi]$ *where* $\bowtie \in \{\leqslant, <, \geqslant, >\}$, $p \in [0,1]$ *and* ψ *is an LTL formula. We say that the probabilistic LTL specification is satisfied in a state* s *of an MDP* $\mathcal{M}$ *if and only if* $Pr^{\sigma}_{\mathcal{M},s}(\psi) \bowtie p$ *for all adversaries* $\sigma \in Adv_{\mathcal{M}}$.

In order to model check an MDP $\mathcal{M}$ and probabilistic LTL specification $\mathtt{P}_{\bowtie p}[\psi]$, we need to compute $Pr^{\min}_{\mathcal{M},s}(\psi)$ if $\bowtie \in \{\geqslant, >\}$ or $Pr^{\max}_{\mathcal{M},s}(\psi)$ if $\bowtie \in \{\leqslant, <\}$. Because every path either satisfies ψ or $\neg\psi$, the problem of computing the minimum probability of satisfying ψ is easily reducible to the computation of the maximum probability of satisfying $\neg\psi$. More precisely, by definition:

$$\begin{aligned} Pr^{\min}_{\mathcal{M},s}(\psi) &= \inf_{\sigma \in Adv_{\mathcal{M}}} Pr^{\sigma}_{\mathcal{M},s}(\{\pi \mid \pi \models \psi\}) \\ &= \inf_{\sigma \in Adv_{\mathcal{M}}} \left(1 - Pr^{\sigma}_{\mathcal{M},s}(\{\pi \mid \pi \not\models \psi\})\right) && \text{since } Pr^{\sigma}_{\mathcal{M},s} \text{ is a probability measure} \\ &= \inf_{\sigma \in Adv_{\mathcal{M}}} \left(1 - Pr^{\sigma}_{\mathcal{M},s}(\{\pi \mid \pi \models \neg\psi\})\right) && \text{by definition of } \models \\ &= 1 - \sup_{\sigma \in Adv_{\mathcal{M}}} Pr^{\sigma}_{\mathcal{M},s}(\{\pi \mid \pi \models \neg\psi\}) && \text{rearranging} \\ &= 1 - Pr^{\max}_{\mathcal{M},s}(\neg\psi) && \text{by definition.} \end{aligned}$$



go
$\alpha_{\mathcal{M}}\backslash\{go\}$
q_0
$\alpha_{\mathcal{M}}\backslash\{go\}$
q_1
go

Fig. 10. A DRA for `F G` *go* with $Acc = \{(\{q_0\}, \{q_1\})\}$ (see Example 15)

Like for probabilistic safety properties in the previous section, LTL model checking is based on the use of automata. However, since the satisfaction of LTL formulas is defined over infinite words, we use ω-automata, rather than finite automata. In particular, as proposed in [1], we use *deterministic Rabin automata.* An alternative, which we do not discuss here, is to use partially determinised Büchi automata [34,35].

Definition 21 (Deterministic Rabin automaton). *A* deterministic Rabin automaton *(DRA) is a tuple* $\mathcal{A} = (Q, \overline{q}, \alpha_{\mathcal{A}}, \delta_{\mathcal{A}}, Acc)$ *of finitely many states* Q*, initial state* $\overline{q} \in Q$*, finite alphabet* $\alpha_{\mathcal{A}}$*, transition function* $\delta_{\mathcal{A}} : Q \times \alpha_{\mathcal{A}} \rightarrow Q$ *and acceptance condition* $Acc = \{(L_i, K_i)\}_{i=1}^{k}$ *where* $k \in \mathbb{N}$ *and* $L_i, K_i \subseteq Q$ *for* $1 \leqslant i \leqslant k$*.*

For a DRA $\mathcal{A} = (Q, \overline{q}, \alpha_{\mathcal{A}}, \delta_{\mathcal{A}}, Acc)$, since the transition function is deterministic and total, for any infinite word $w = a_0 a_1 a_2 \ldots$ over $\alpha_{\mathcal{A}}$ there is a corresponding unique path $\overline{q} \xrightarrow{a_0} q_1 \xrightarrow{a_1} q_2 \xrightarrow{a_2} \cdots$ of $\mathcal{A}$ (where the path of $\mathcal{A}$ is defined as for DFAs). Using this fact, we say that $\mathcal{A}$ *accepts* an infinite word w if the corresponding path contains *finitely* many states from L_i and *infinitely* many from K_i for some $1 \leqslant i \leqslant k$. The language $\mathcal{L}(\mathcal{A})$ of the DRA $\mathcal{A}$ is given by the set of infinite words that the automaton accepts.

Now, for any LTL formula ψ, we can construct a DRA, say $\mathcal{A}_\psi$, over $\alpha_{\mathcal{M}}$ that accepts precisely the words satisfying ψ, i.e. for any infinite word w:

$$w \models \psi \quad \Longleftrightarrow \quad w \in \mathcal{L}(\mathcal{A}_\psi)$$

The construction of the DRA $\mathcal{A}_\psi$ from the formula ψ is beyond the scope of this tutorial; for details see e.g. [85,36,11].

In fact, the set of properties that can be captured by a DRA is a strict superset of those expressible as LTL formulas, known as *ω-regular properties.* The same class of properties can also be captured with nondeterministic Büchi automata. However, the model checking process for MDPs requires *deterministic* automata and deterministic Büchi automata are not sufficiently expressive (e.g. the LTL formula `F G` a cannot be represented by a deterministic Büchi automaton).

Example 15. Consider again the running example of Figure 2, together with the DRA $\mathcal{A} = (Q, q_0, \alpha_{\mathcal{M}}, \delta_{\mathcal{M}}, Acc)$ given in Figure 10, i.e. $Q = \{q_0, q_1\}$, $\delta_{\mathcal{A}}(q, go) = q_1$ and $\delta_{\mathcal{A}}(q, a) = q_0$ for all $q \in Q$ and $a \in \alpha_{\mathcal{M}} \backslash \{go\}$, and Acc is a singleton set containing $(\{q_0\}, \{q_1\})$. The property specified by the automaton is "eventually we will only take the action *go*", which is equivalent to the LTL formula `F G` *go*. This holds with probability 0 under all adversaries, since, for any adversary, almost all paths eventually reach and stay in either s_2 or s_3. ■

7.3 Model Checking LTL and ω-Regular Properties

We now describe the process of computing the maximum probabilities, in an MDP $\mathcal{M} = (S, \overline{s}, \alpha_{\mathcal{M}}, \delta_{\mathcal{M}}, L)$, for an ω-regular property given in the form of a DRA $\mathcal{A}$. As mentioned earlier, this subsumes the problem of computing the probabilities for an LTL formula. This is done by constructing the *product* of $\mathcal{M}$ and $\mathcal{A}$, and then identifying *accepting end components.* To ease presentation, for the remainder of the section we omit the labelling function from MDPs.

Definition 22 (MDP-DRA product). *The* product $\mathcal{M}{\otimes}\mathcal{A}$ *of an MDP* $\mathcal{M} = (S, \overline{s}, \alpha_{\mathcal{M}}, \delta_{\mathcal{M}})$ *and DRA* $\mathcal{A} = (Q, \overline{q}, \alpha_{\mathcal{M}}, \delta_{\mathcal{A}}, \{(L_i, K_i)\}_{i=1}^{k})$ *is given by the MDP* $(S{\times}Q, (\overline{s}, \overline{q}), \alpha_{\mathcal{M}}, \delta_{\mathcal{M}\otimes\mathcal{A}})$ *where, for any* $(s, q) \in S{\times}Q$ *and* $a \in \alpha_{\mathcal{M}}$*:*

– $\delta_{\mathcal{M}\otimes\mathcal{A}}((s,q),a) = \begin{cases} \delta_{\mathcal{M}}(s,a){\times}[\delta_{A}(q,a){\mapsto}1] & \text{if } a \in A(s) \\ \text{undefined} & \text{otherwise.} \end{cases}$

For an MDP $\mathcal{M}$ and DRA $\mathcal{A}$ with acceptance condition $\{(L_i, K_i)\}_{i=1}^{k}$, an *accepting path* of $\mathcal{M}{\otimes}\mathcal{A}$ is an infinite path such that, when projecting its states onto Q, there are *finitely* many states from L_i and *infinitely* many from K_i for some $1{\leqslant}i{\leqslant}k$. Furthermore, an *accepting EC* (recall the definition of end components from Definition 6) of $\mathcal{M}{\otimes}\mathcal{A}$ is an EC (S', δ') for which there exists an $1{\leqslant}i{\leqslant}k$ such that the set of states S', when projected onto Q, contains some state from K_i, but no states from L_i. A key property of the product $\mathcal{M}{\otimes}\mathcal{A}$ is that, for every state s and adversary σ in $\mathcal{M}$, there is an adversary σ' in $\mathcal{M}{\otimes}\mathcal{A}$ such that:

$$\begin{aligned} & Pr^{\sigma}_{\mathcal{M},s}(\{\pi \in IPath_{\mathcal{M},s} \mid tr(\pi) \in \mathcal{L}(\mathcal{A})\}) \\ & \quad = Pr^{\sigma'}_{\mathcal{M}\otimes\mathcal{A},(s,\overline{q})}(\{\pi' \in IPath_{M\otimes\mathcal{A},(s,\overline{q})} \mid \pi' \text{ is an accepting path}\}) \end{aligned}$$

and vice versa. Using this property, together with properties of end components [1], we can further show that the following proposition holds.

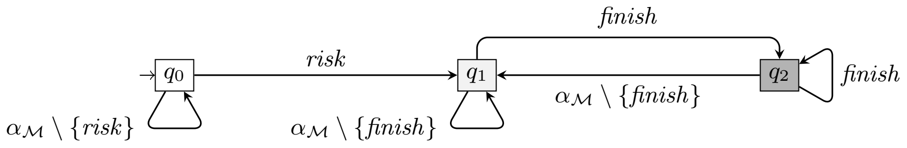
Proposition 2. *For any MDP* $\mathcal{M} = (S, \overline{s}, \alpha_{\mathcal{M}}, \delta_{\mathcal{M}})$*, state* $s \in S$ *and DRA* $\mathcal{A} = (Q, \overline{q}, \alpha_{\mathcal{M}}, \delta_{\mathcal{A}}, \{(L_i, K_i)\}_{i=1}^{k})$*, we have:*

$$Pr^{\max}_{\mathcal{M},s}(\{\pi \in IPath_{\mathcal{M},s} \mid tr(\pi) \in \mathcal{L}(\mathcal{A})\}) = Pr^{\max}_{\mathcal{M}\otimes\mathcal{A},(s,\overline{q})}(reach(T))$$

where $(s', q') \in T$ *if and only if* (s', q') *appears in some accepting EC of* $\mathcal{M}{\otimes}\mathcal{A}$*.*

We have therefore reduced the problem of model checking LTL and ω-regular properties to the detection of end components in a product MDP and the computation of maximum reachability probabilities. The latter is covered in Section 4, while [1,11] describes computation of end components. Furthermore, [11] shows how to optimise model checking by considering only *maximal* ECs.

For a given MDP $\mathcal{M}$ and DRA $\mathcal{A} = (Q, \overline{q}, \alpha_{\mathcal{M}}, \delta_{\mathcal{A}}, \{(L_i, K_i)\}_{i=1}^{k})$, suppose that we have obtained (for example through the techniques of Section 4), a memoryless adversary $\sigma^{\max}$ that maximises the probability of reaching a state in an accepting EC of the product. We can then construct a finite-memory adversary for $\mathcal{M}$ that maximises the probability of satisfying the corresponding



finish
risk
q_0
q_1
q_2
finish
$\alpha_{\mathcal{M}} \setminus \{finish\}$
$\alpha_{\mathcal{M}} \setminus \{risk\}$
$\alpha_{\mathcal{M}} \setminus \{finish\}$

Fig. 11. A DRA for $(\mathtt{F}\ \mathtt{G}\ \mathit{finish}) \wedge (\mathtt{F}\ \mathit{risk})$ with $Acc = \{(\{q_1\}, \{q_2\})\}$ (see Example 16)

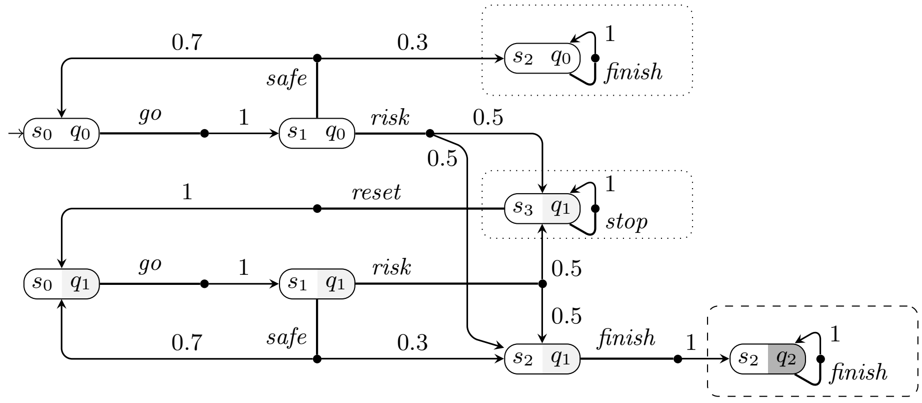


Fig. 12. The product MDP $\mathcal{M}\otimes\mathcal{A}$ for Example 16

ω-regular property. This adversary is specified by the tuple $(Q, \overline{q}, \sigma_u, \sigma_s)$ (see Definition 9) where $\sigma_u(q,a,s){=}\delta_{\mathcal{A}}(q,a)$ and $\sigma_s(q,s){=}\sigma^{\max}((s,q))$ for all $q \in Q$, $s \in S$ and $a \in \alpha_M$.

Example 16. Let us return to the running example of Figure 2 and compute the maximum probability that $\psi = (\mathtt{F}\ \mathtt{G}\ \mathit{finish}) \wedge (\mathtt{F}\ \mathit{risk})$ is satisfied. Intuitively, this means maximising the probability of eventually reaching and staying in the "finished" state, while at the same time taking a risky decision at least once. A DRA $\mathcal{A}$ that accepts precisely the words satisfying ψ is depicted in Figure 11.

To compute the maximum probability, we first construct the product $\mathcal{M}\otimes\mathcal{A}$. Restricting the MDP to states that are reachable from (s_0, q_0) yields the MDP in Figure 12. It has three end components, which are denoted in the figure as follows: the (single) accepting end component is represented by a dashed box, the remaining two by dotted boxes. From Proposition 2, $Pr^{\max}_{\mathcal{M},s_0}(\psi)$ equals $Pr^{\max}_{\mathcal{M}\otimes\mathcal{A},(s_0,q_0)}(\mathit{reach}(\{(s_2,q_2)\}))$, which we now proceed to compute using the methods from Section 4. First, using Algorithm 3 and Algorithm 4, we find the sets $S^0_{\max}{=}\{(s_2,q_0)\}$ and $S^1_{\max}{=}S\backslash S^0_{\max}$. This gives us the probabilities for all states and, since $(s_0,q_0) \in S^1_{\max}$, we have $Pr^{\max}_{\mathcal{M},s_0}(\psi){=}1$.

An example optimal adversary σ' in $\mathcal{M}\otimes\mathcal{A}$ is the one that satisfies $\sigma'(\rho) = [\mathit{reset}{\mapsto}1]$ for all ρ ending in (s_3,q_1), and $\sigma'(\rho) = [\mathit{risk}{\mapsto}1]$ for all ρ ending in (s_1,q) for any q. This in fact transfers directly to a memoryless optimal adversary σ in $\mathcal{M}$ satisfying $\sigma(s_3) = [\mathit{reset}{\mapsto}1]$ and $\sigma(s_1) = [\mathit{risk}{\mapsto}1]$. ■

Complexity. We now discuss the complexity of the algorithm above. Starting with LTL formula ψ and MDP $\mathcal{M}$, we first construct a DRA $\mathcal{A}_\psi$. In the worst case, the size $|\mathcal{A}_\psi|$ (number of states) of the smallest DRA $\mathcal{A}_\psi$ can be doubly exponential in $|\psi|$, and the time complexity of the computation is doubly exponential as well. In practice, though, both ψ and $\mathcal{A}_\psi$ are much smaller than $\mathcal{M}$.

After computing $\mathcal{A}_\psi$, we construct the product $\mathcal{M}\otimes\mathcal{A}_\psi$, which can be done in time polynomial in $|\mathcal{M}|$ and $|\mathcal{A}_\psi|$. Then, we identify the end components and compute reachability probabilities, both in time polynomial in the size of the product. Overall, we get that the algorithm runs in time polynomial in $|\mathcal{M}|$ and doubly exponential in $|\psi|$.

8 Multi-objective Probabilistic Model Checking

In this section, we consider *multi-objective* verification techniques for MDPs [40]. These permit the analysis of trade-offs between several linear-time objectives, for example "the probability of reaching a good state is at least 0.98 *and*, with probability at most 0.3, it will be reached in more than 10 steps". These techniques will also be used, in Section 9, to perform compositional verification of MDPs. We begin with the problem of multiple probabilistic reachability objectives, and then describe how to generalise this to multiple ω-regular objectives.

Definition 23 (Multi-objective reachability query). *For an MDP $\mathcal{M}$, target sets $T_1,\ldots,T_n \subseteq S$, relational operators $\rhd_1,\ldots,\rhd_n \in \{>,\geqslant\}$ and bounds $p_1 \ldots, p_n \in [0,1]$, a* multi-objective reachability query *asks whether there exists an adversary $\sigma \in Adv_{\mathcal{M}}$ such that $Pr^{\sigma}_{\mathcal{M},\overline{s}}(reach(T_i)) \rhd_i p_i$ for all $1\leqslant i\leqslant n$.*

Observe two key differences between this type of query and the verification problems we have considered so far in this tutorial: firstly, the quantification over adversaries is *existential*, not universal; and secondly, we are concerned only with the probability from the initial state $\overline{s}$ of $\mathcal{M}$.

For technical reasons, we assume that all states in the target sets T_i are absorbing, i.e. the only available transitions are self-loops. This restriction is lifted in the generalised version of the problem that we discuss subsequently. Letting $T = \cup_{i=1}^{n} T_i$, we also compute the states from which no $s' \in T$ is reachable. These states can be identified by computing the set $S^0_{\max}$ from Section 4. Supposing that $\overline{s} \notin S^0_{\max} \cup T$ (as otherwise the solution is trivial), it can be shown [40] that there exists an adversary σ such that $Pr^{\sigma}_{\mathcal{M},\overline{s}}(reach(T_i)) \rhd_i p_i$ for all $1\leqslant i\leqslant n$ if and only if the set of inequalities $L(\mathcal{M})$ in Figure 13 has a solution.

Furthermore, it turns out that a solution to the inequalities $L(\mathcal{M})$ yields a *memoryless randomised* adversary σ under which $Pr^{\sigma}_{\mathcal{M},\overline{s}}(reach(T_i)) \rhd_i p_i$ for all $1\leqslant i\leqslant n$. The adversary σ is defined by $\sigma(s)(a) = y_{(s,a)}/(\sum_{a'\in A(s)} y_{(s,a')})$ whenever the denominator is non-zero, and arbitrarily for all other states s (such states are not reachable from $\overline{s}$ under σ). The intuition behind the solution to $L(\mathcal{M})$ is that, the variables $y_{(s,a)}$ represent the expected number of times, under adversary σ, that a is taken in s, when starting in $\overline{s}$. This is ensured by the equations on the first and last lines of $L(\mathcal{M})$, which intuitively state that the

$$\begin{array}{rll} \mathit{in}(s) + \sum_{s' \in U} \sum_{a \in A(s')} \delta(s',a)(s) \cdot y_{(s',a)} = & \sum_{a \in A(s)} y_{(s,a)} & \text{for all } s \in U \\ \sum_{s \in T_i} \sum_{s' \in U} \sum_{a \in A(s')} \delta(s',a)(s) \cdot y_{(s',a)} \rhd_i & p_i & \text{for all } 1 \leqslant i \leqslant n \\ y_{(s,a)} \geqslant & 0 & \text{for all } s \in U \text{ and } a \in A(s) \end{array}$$

where $U = S \backslash (T \cup S^0_{\max})$ and $\mathit{in}(s) = 1$ if $s = \overline{s}$ and equals 0 otherwise.

Fig. 13. The linear inequalities $L(\mathcal{M})$ for multi-objective reachability

number of times we enter a state must be equal to the number of times we leave it. The equations on the second line capture the conditions imposed on the probabilities of reaching each target set T_i. Since the target states are absorbing and their outgoing self-loops are omitted from $L(\mathcal{M})$, the expected number of times to enter a target state equals the probability of reaching it.

Multi-objective LTL Queries. We now generalise these multi-objective techniques to the case of ω-regular properties. For simplicity, our presentation will be in terms of LTL formulas.

Definition 24 (Multi-objective LTL query). *A* multi-objective LTL query *is a formula generated by the following grammar:*

$$\theta \ ::= \ \mathtt{P}_{\bowtie p}[\psi] \mid \neg\theta \mid \theta \wedge \theta$$

where ψ is an LTL formula, $\bowtie \in \{<, \leqslant, \geqslant, >\}$, and $p \in [0,1]$.

As before, we allow the use of connectives $\theta_1 \vee \theta_2$ and $\theta_1 \rightarrow \theta_2$ as abbreviations for $\neg(\neg\theta_1 \wedge \neg\theta_2)$ and $\neg\theta_1 \vee \theta_2$, respectively. The semantics of multi-objective LTL queries is defined with respect to both a state s and a specific adversary σ. As above, the verification problem is *existential*: we need to decide, given an MDP $\mathcal{M}$ and multi-objective LTL query θ, whether θ is satisfied in the initial state $\overline{s}$ *for some* adversary σ of $\mathcal{M}$. Formally, the semantics is defined as follows.

Definition 25 (Multi-objective LTL semantics). *Let $\mathcal{M}{=}(S, \overline{s}, \alpha_{\mathcal{M}}, \delta_{\mathcal{M}})$ be an MDP, $s \in S$ a state and σ an adversary of $\mathcal{M}$. The satisfaction of a multi-objective LTL query θ is defined inductively by:*

$$\begin{array}{lcl} \sigma, s \models \mathtt{P}_{\bowtie p}[\psi] & \iff & \mathit{Pr}^{\sigma}_{\mathcal{M},s}(\psi) \bowtie p \\ \sigma, s \models \neg\theta & \iff & \sigma, s \not\models \theta \\ \sigma, s \models \theta_1 \wedge \theta_2 & \iff & \sigma, s \models \theta_1 \textit{ and } \sigma, s \models \theta_2 \end{array}$$

The result of the query θ is `true` *in $\mathcal{M}$ if and only if $\sigma, \overline{s} \models \theta$ for some $\sigma \in \mathit{Adv}_{\mathcal{M}}$.*

Note the following difference in the semantics from PCTL (see Section 6) in the way it quantifies over adversaries. In the PCTL semantics, we quantify over adversaries every time we evaluate a formula $\mathtt{P}_{\bowtie p}[\psi]$, while in the case of multi-objective queries we evaluate the whole formula under one fixed adversary.

As the first step towards the solution of multi-objective problems, we employ well-established results of propositional logic and transform a given multi-objective query θ to an equivalent query θ' in disjunctive normal form, i.e. θ' is a disjunction of conjunctions of literals of the form $\mathtt{P}_{\bowtie p}[\psi]$ or $\neg\mathtt{P}_{\bowtie p}[\psi]$. We can further remove negations by replacing $\mathtt{P}_{\bowtie p}[\psi]$ with $\mathtt{P}_{\bar{\bowtie} p}[\psi]$ where $\bar{\bowtie}$ is chosen appropriately, e.g. $\bar{\leqslant}{=}{>}$. Finally, we can ensure that no comparison operators $\leqslant$ or $<$ occur in the formula, by replacing $\mathtt{P}_{<p}[\psi]$ with $\mathtt{P}_{>1-p}[\neg\psi]$, and $\mathtt{P}_{\leqslant p}[\psi]$ with $\mathtt{P}_{\geqslant 1-p}[\neg\psi]$. We obtain a formula which is a disjunction of clauses of the form:

$$\mathtt{P}_{\rhd_1 p_1}[\psi_1] \wedge \ldots \wedge \mathtt{P}_{\rhd_n p_n}[\psi_n]$$

where each $\rhd_i$ is $\geqslant$ or $>$. We can then analyse each clause separately, concluding that the formula is true if and only if at least one clause is true. For this reason, for the remainder of this section, we can assume that the multi-objective query θ is a conjunction of propositions $\mathtt{P}_{\rhd_i p_i}[\psi_i]$ for $\rhd_i \in \{\geqslant, >\}$.

The next step of the solution is similar to standard (single-objective) LTL model checking (see Section 7): we convert each LTL formula ψ_i to a DRA $\mathcal{A}_i$ such that $w \models \psi_i \iff w \in \mathcal{L}(\mathcal{A}_i)$ and then construct the product MDP $\mathcal{M}' = (\cdots((\mathcal{M}\otimes\mathcal{A}_1)\otimes\mathcal{A}_2)\cdots)\otimes\mathcal{A}_n$. Next, for each subset $X \subseteq \{1,\ldots,n\}$ we identify the end components of $\mathcal{M}'$ that are accepting for all $\mathcal{A} \in \{\mathcal{A}_i \mid i \in X\}$. As in Section 7, an end component is accepting for $\mathcal{A}$ if its acceptance condition contains some (L_j, K_j) such that the states of the EC, when projected onto the states of $\mathcal{A}$, contain some state from K_j, but no states from L_j.

It can then be shown that the multi-objective LTL query θ is satisfied for some adversary of $\mathcal{M}$ if and only if there exists an adversary of $\mathcal{M}'$ which ensures that, from initial state $(\overline{s}, \overline{q}_1, \ldots, \overline{q}_n)$ of $\mathcal{M}'$, the probability of eventually reaching and staying in end components that are accepting for $\mathcal{A}_i$ satisfies $\rhd_i\, p_i$ for $1{\leqslant}i{\leqslant}n$. To determine whether such an adversary exists, we reduce the problem to a multi-objective reachability problem, which can then be solved via the inequalities presented earlier in Figure 13. The reduction involves the construction, based on $\mathcal{M}' = (S', \overline{s}', \alpha_{\mathcal{M}}, \delta'_{\mathcal{M}})$, of another MDP $\mathcal{M}'' = (S'', \overline{s}'', \alpha''_{\mathcal{M}}, \delta''_{\mathcal{M}})$ where:

- $S'' = S' \cup \{s_X \mid X{\subseteq}\{1,\ldots,n\}\}$ and $\overline{s}''{=}\overline{s}'$;
- $\alpha''_{\mathcal{M}} = \alpha_{\mathcal{M}} \cup \{a_X \mid X \subseteq \{1,\ldots,n\}\}$;
- $\delta''_{\mathcal{M}}$ is created from the probabilistic transition function of $\mathcal{M}'$ by adding transitions $\delta''_{\mathcal{M}}(s, a_X) = [s_X{\mapsto}1]$ for every s and $X \subseteq \{1,\ldots,n\}$ such that s is in an end component that is accepting for all $\mathcal{A} \in \{\mathcal{A}_i \mid i \in X\}$.

The following statement captures the correspondence between $\mathcal{M}'$ and $\mathcal{M}''$.

Proposition 3. *For any $1{\leqslant}i{\leqslant}n$, there is an adversary of $\mathcal{M}'$ under which, from $\overline{s}'$, the probability of reaching and staying in end components that are accepting for $\mathcal{A}_i$ satisfies $\rhd_i\, p_i$, if and only if there is an adversary of $\mathcal{M}''$ under which, from $\overline{s}''$, set $T_i{=}\{s_X \mid X \subseteq \{1,\ldots,n\} \wedge i \in X\}$ is reached with probability $\rhd_i\, p_i$.*

Thus, we have a method to check a multi-objective LTL query on an MDP $\mathcal{M}$, via multi-objective reachability on $\mathcal{M}''$. We conclude by describing how, when a satisfying adversary of $\mathcal{M}$ is shown to exist, it can be constructed from the adversary σ'' of $\mathcal{M}''$ obtained via multi-objective reachability. First, we construct the adversary σ' of $\mathcal{M}'$ which follows the decisions of σ'' except that, instead of taking actions a_X, it "switches" its mode and starts mimicking an adversary that stays in end components that are accepting for all $\mathcal{A} \in \{\mathcal{A}_i \mid i \in X\}$ and visits all its states infinitely often. The adversary σ can then be constructed from σ' using the techniques presented in Section 7.

Example 17. Consider the MDP $\mathcal{M}$ of Figure 14(a) and the multi-objective query $\mathtt{P}_{\geqslant 0.6}[\mathtt{F}\ b_1] \wedge \mathtt{P}_{\geqslant 0.3}[\mathtt{G}\ b_2]$. The formula is already a conjunction of propositions and contains only the comparison operator $\geqslant$, so we proceed with construction of the equivalent DRAs $\mathcal{A}_1$ and $\mathcal{A}_2$ for the formulas $\mathtt{F}\ b_1$ and $\mathtt{G}\ b_2$, respectively. The automata are depicted in Figures 14(b)–(c) and the accepting tuples are $Acc_1 = \{(\varnothing, \{q_1\})\}$ and $Acc_2 = \{(\varnothing, \{q_2\})\}$.

We next construct the product MDP $\mathcal{M}'{=}\mathcal{M}{\otimes}\mathcal{A}_1{\otimes}\mathcal{A}_2 = (S', \overline{s}', \alpha_{\mathcal{M}}, \delta'_{\mathcal{M}})$, where $S'{=}S{\times}\{q_0, q_1\}{\times}\{q_2, q_3\}$, $\overline{s}'{=}(t_0, q_0, q_2)$, and the structure of the part of $\mathcal{M}'$ reachable from $\overline{s}'$ is exactly the same as the structure of $\mathcal{M}$, except that t_0, t_1 and t_2 are replaced with (t_0, q_0, q_2), (t_1, q_1, q_3) and (t_2, q_0, q_2), respectively. The MDP has 2 end components $C_1 = (\{(t_1, q_1, q_3)\}, \delta_1)$ and $C_2 = (\{(t_0, q_0, q_2)(t_2, q_0, q_2)\}, \delta_2)$ where δ_1 and δ_2 are uniquely determined by the set of states contained in the end component. We next construct the MDP $\mathcal{M}''$ according to the definition introduced earlier on page 93. The part of $\mathcal{M}''$ reachable from (t_0, q_0, q_2) is depicted in Figure 15. Our target sets for multi-objective reachability in $\mathcal{M}''$ are $T_1 = \{s_{\{1\}}\}$ and $T_2 = \{s_{\{2\}}\}$. We then get the following set of inequalities $L(\mathcal{M}'')$:

$$\begin{aligned}
1 + y_{((t_2,q_0,q_2),b_2)} + 0.5{\cdot}y_{((t_0,q_0,q_2),b_2)} &= y_{((t_0,q_0,q_2),b_1)} + y_{((t_0,q_0,q_2),b_2)} + y_{((t_0,q_0,q_2),a_{\{2\}})} \\
y_{((t_0,q_0,q_2),b_1)} + y_{((t_1,q_1,q_3),b_1)} &= y_{((t_1,q_1,q_3),b_1)} + y_{((t_1,q_1,q_3),a_{\{1\}})} \\
0.5{\cdot}y_{((t_0,q_0,q_2),b_2)} &= y_{((t_2,q_0,q_2),b_2)} + y_{((t_2,q_0,q_2),a_{\{2\}})} \\
y_{((t_1,q_1,q_3),a_{\{1\}})} &\geqslant 0.6 \\
y_{((t_0,q_0,q_2),a_{\{2\}})} + y_{((t_2,q_0,q_2),a_{\{2\}})} &\geqslant 0.3
\end{aligned}$$

These equations *do* have (infinitely many) solutions and we can pick an arbitrary one, e.g. the solution with non-zero values:

$$\begin{aligned}
y_{((t_0,q_0,q_2),b_1)} &= 0.6 \\
y_{((t_0,q_0,q_2),b_2)} &= 0.8 \\
y_{((t_1,q_1,q_3),a_{\{1\}})} &= 0.6 \\
y_{((t_2,q_0,q_2),a_{\{2\}})} &= 0.4\,.
\end{aligned}$$

This gives adversary σ'' which, in (t_0, q_0, q_2), picks b_1 or b_2 with probability $0.6/(0.6{+}0.8){=}\frac{3}{7}$ or $0.8/(0.6{+}0.8){=}\frac{4}{7}$, respectively, and in (t_1, q_1, q_3) and (t_2, q_0, q_2) chooses $a_{\{1\}}$ and $a_{\{2\}}$ deterministically. The induced adversary σ' of $\mathcal{M}'$ is both randomised and history dependent: if only (t_0, q_0, q_2) is in the history, it selects $[b_1{\mapsto}\frac{3}{7}, b_2{\mapsto}\frac{4}{7}]$; for all other paths ending in (t_0, q_0, q_2), it selects $[b_2{\mapsto}1]$. The adversary σ' maps naturally to an adversary σ of $\mathcal{M}$ that satisfies the query $\mathtt{P}_{\geqslant 0.6}[\mathtt{F}\ b_1] \wedge \mathtt{P}_{\geqslant 0.3}[\mathtt{G}\ b_2]$. ■

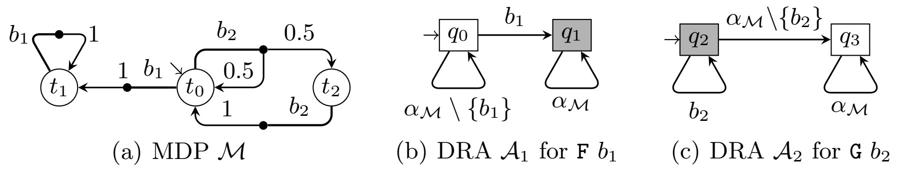

(a) MDP $\mathcal{M}$
(b) DRA $\mathcal{A}_1$ for F b_1
(c) DRA $\mathcal{A}_2$ for G b_2

Fig. 14. The MDP and deterministic Rabin automata for Example 17

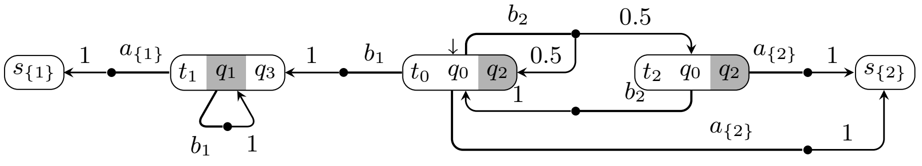


Fig. 15. The MDP $\mathcal{M}''$, built from $\mathcal{M}' = \mathcal{M} \otimes \mathcal{A}_1 \otimes \mathcal{A}_2$, for Example 17

Quantitative Approaches. The techniques presented in this section can also be generalised in several other ways. Firstly, like for the other verification problems in this chapter, we can consider *quantitative* approaches to multi-objective model checking [69,45]. We can define *numerical* queries, which may be more useful than existential ones in practice. These optimise one objective, subject to constraints on several others. Formally we have the following definition.

Definition 26 (Numerical multi-objective query). *For a PA $\mathcal{M}$ with initial state $\overline{s}$, multi-objective LTL query θ and LTL formula ψ, a* (maximising) numerical multi-objective query *is to find the following value:*

$$Pr^{\max}_{\mathcal{M},\overline{s}}(\psi \,|\, \theta) \overset{\text{def}}{=} \sup\{Pr^{\sigma}_{\mathcal{M},\overline{s}}(\psi) \mid \sigma \in Adv_{\mathcal{M}} \wedge \sigma, \overline{s} \models \theta\}\,.$$

If the property θ is not satisfied by any adversary of $\mathcal{M}$, the query returns $\perp$. A minimising *numerical multi-objective query is defined similarly.*

Numerical queries can solved at essentially the same cost as existential ones. This is done by adding an objective function to the set of linear inequalities and solving a linear program instead. Multi-objective queries can be further extended by integrating reward-based properties similar to those in Section 5; see [45].

Complexity. Consider a multi-objective LTL query θ for an MDP $\mathcal{M}$. Converting the query to disjunctive normal form can be done in time exponential in the number of connectives of θ, yielding at most exponentially many clauses, each containing at most polynomially many literals. The formula in disjunctive normal form, with negations and operators $<$ and $\leqslant$ removed, contains at most k different LTL formulas, where k is polynomial in the size of θ. We convert each LTL formula to a DRA in time doubly exponential in its size.

For a single clause containing n LTL formulas, we then build the product $\mathcal{M}'$ of $\mathcal{M}$ and the constructed DRAs. This product is polynomial in the size of $\mathcal{M}$,

n and the size of the DRAs. Identifying accepting end components and then constructing the MDP $\mathcal{M}''$ both require time polynomial in the size of $\mathcal{M}'$ and exponential in n. The set of equations is constructed in time polynomial in $\mathcal{M}''$, and solved in polynomial time using the techniques for linear programming. The whole procedure thus runs in the time doubly exponential in the size of θ, and polynomial in the size of $\mathcal{M}$.

Controller Synthesis. The problems discussed in this chapter concern the generation of MDP adversaries that satisfy certain formally-specified properties. This is often referred to as *controller synthesis*, and can be generalised in several ways. We can, for example, return to the temporal logic PCTL defined in Section 6 and consider an alternative semantics for the logic. Let us define validity of a PCTL formula under a fixed adversary σ by stipulating $Adv{=}\{\sigma\}$ in the semantics presented in Definition 13. The problem is to determine whether there exists a σ under which the given formula is true.

This problem has also been studied [9,21,20], and—perhaps surprisingly—it is fundamentally different from the problem in which Adv is the set of all adversaries. In particular, answering the question whether there is a satisfying adversary is undecidable. It becomes decidable when we restrict to qualitative case (i.e. when the probability bounds in the formula are taken from the set $\{0, 1\}$), but even then a satisfying adversary may require infinite memory.

9 Compositional Probabilistic Model Checking

In this section, we discuss *compositional* approaches to probabilistic model checking of MDPs, in particular illustrating an *assume-guarantee* framework [69].

9.1 Probabilistic Automata and Parallel Composition

System designs often comprise multiple components operating in parallel. When these components exhibit stochastic behaviour, MDPs are a natural formalism to model the system since nondeterminism can be used to capture concurrency between the components. In fact, for the purposes of compositional modelling and analysis, it is preferable to use probabilistic automata (PAs) [80,81], which are a (slight) generalisation of MDPs. The essential difference is that a state of a PA can have more than one outgoing transition with the same action label.

Definition 27 (Probabilistic automaton). *A* probabilistic automaton *(PA) is a tuple $\mathcal{M}{=}(S, \overline{s}, \alpha_{\mathcal{M}}, \delta_{\mathcal{M}}, L)$, where S is a finite set of states, $\overline{s} \in S$ is an initial state, $\alpha_{\mathcal{M}}$ is a finite alphabet, $\delta_{\mathcal{M}} \subseteq S{\times}(\alpha_{\mathcal{M}}{\cup}\{\tau\}){\times}Dist(S)$ is a finite probabilistic transition relation and $L : S \to 2^{AP}$ is a labelling function mapping each state to a set of atomic propositions taken from a set AP.*

Notice that $\delta_{\mathcal{M}}$ is now a relation, unlike in Definition 4 (see page 57) where it is a function. Observe also that we allow transitions to be labelled with a special τ action, representing "silent" transitions that are internal to some component. We introduce additional notation and use $s \xrightarrow{a} \mu$ to denote that $(s, a, \mu) \in \delta_{\mathcal{M}}$.

Basic notions such as paths, traces and adversaries of MDPs (see Section 3) need slight modifications for PAs, but this is straightforward. In the case of traces, for example, we omit τ actions, following the intuition that these actions are "silent". A technical detail required by the compositional approach of [69] is the use of *partial adversaries*, which can opt to (with some probability) take none of the available choices and remain in the current state. However, model checking of *probabilistic safety properties*, on which the compositional techniques described here are based, is unaffected by this distinction (intuitively, this is because remaining in a state can only decrease the probability of satisfying a safety property). The definition of a PA-DFA product and the process for model checking probabilistic safety properties are also essentially the same as the MDP case; see [69] for details.

We now introduce a few additional concepts required for *compositional* modelling and analysis of PAs.

Definition 28 (Parallel composition of PAs). *If* $\mathcal{M}_i = (S_i, \overline{s}_i, \alpha_{\mathcal{M}_i}, \delta_{\mathcal{M}_i}, L_i)$ *are PAs for* $i{=}1,2$*, then their* parallel composition*, denoted* $\mathcal{M}_1 \| \mathcal{M}_2$*, is given by the PA* $(S_1 \times S_2, (\overline{s}_1, \overline{s}_2), \alpha_{\mathcal{M}_1} \cup \alpha_{\mathcal{M}_2}, \delta_{\mathcal{M}_1 \| \mathcal{M}_2}, L)$ *where* $\delta_{\mathcal{M}_1 \| \mathcal{M}_2}$ *is defined such that* $(s_1, s_2) \xrightarrow{a} \mu_1 \times \mu_2$ *if and only if one of the following holds:*

- $s_1 \xrightarrow{a} \mu_1$*,* $s_2 \xrightarrow{a} \mu_2$ *and* $a \in \alpha_{\mathcal{M}_1} \cap \alpha_{\mathcal{M}_2}$
- $s_1 \xrightarrow{a} \mu_1$*,* $\mu_2 = [s_2 \mapsto 1]$ *and* $a \in \alpha_{\mathcal{M}_1} \backslash \alpha_{\mathcal{M}_2}$
- $\mu_1 = [s_1 \mapsto 1]$*,* $s_2 \xrightarrow{a} \mu_2$ *and* $a \in \alpha_{\mathcal{M}_2} \backslash \alpha_{\mathcal{M}_1}$

and $L(s_1, s_2) = L_1(s_1) \cup L_2(s_2)$*.*

This form of parallel composition [80,81], which allows multi-way synchronisation over the same action by several components, is in a similar style to the scheme used in the process algebra CSP [79] and is also used in PRISM [56]. By default, we assume that $\mathcal{M}_1$ and $\mathcal{M}_2$ synchronise over all common actions. However, this can easily be generalised to incorporate more flexible definitions of synchronisation, as well as operators to hide and rename action labels.

Definition 29 (Alphabet extension of PA). *For a PA* $\mathcal{M}{=}(S, \overline{s}, \alpha_{\mathcal{M}}, \delta_{\mathcal{M}}, L)$ *and set of actions* α*, we* extend *$\mathcal{M}$'s alphabet to α, denoted* $\mathcal{M}[\alpha]$*, as follows:* $\mathcal{M}[\alpha]{=}(S, \overline{s}, \alpha_{\mathcal{M}} {\cup} \alpha, \delta_{\mathcal{M}[\alpha]}, L)$ *where* $\delta_{\mathcal{M}[\alpha]} {=} \delta_{\mathcal{M}} {\cup} \{(s, a, [s {\mapsto} 1]) \mid s {\in} S {\wedge} a {\in} \alpha \backslash \alpha_{\mathcal{M}}\}$*.*

Example 18. Figure 16 shows two PAs used to model a system comprising a machine ($\mathcal{M}_2$) and a controller ($\mathcal{M}_1$) that is responsible for powering it down. When $\mathcal{M}_1$ *detect*s a problem, it should send two messages: *warn* and then *shutdown* to $\mathcal{M}_2$. However, with probability 0.2, it fails to transmit *warn*. If $\mathcal{M}_2$ does not receive the *warn* message before the *shutdown* message, there is a 10% chance it will *fail* to shut down correctly. Figure 16(c) also shows the DFA $\mathcal{A}_G^{err}$ for a safety property Φ_G "action *fail* never occurs". On the parallel composition $\mathcal{M}_1 \| \mathcal{M}_2$, we can compute $Pr^{\min}_{\mathcal{M}_1 \| \mathcal{M}_2, (s_0, t_0)}(\Phi_G) = 1 - 0.2 \cdot 0.1 = 0.98$. Thus, the initial state of $\mathcal{M}_1 \| \mathcal{M}_2$ satisfies the probabilistic safety property $\mathtt{P}_{\geqslant 0.98}[\Phi_G]$. ■

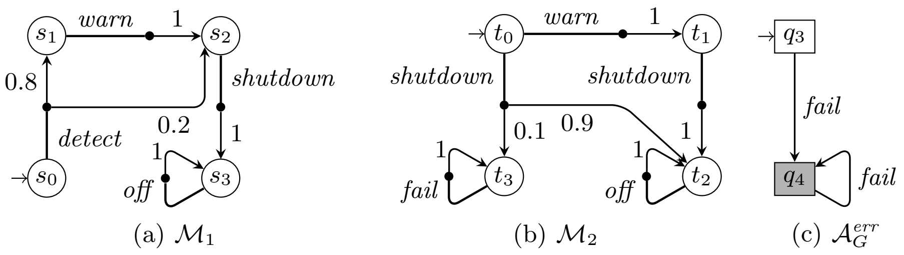


Fig. 16. Two PAs $\mathcal{M}_1, \mathcal{M}_2$ and a DFA $\mathcal{A}_G^{err}$ for a safety property Φ_G

9.2 Assume-Guarantee Verification

We now describe the approach for *compositional* verification of probabilistic automata presented in [69]. This is based on the popular *assume-guarantee* paradigm, in which components of a system are verified separately, under *assumptions* about their environment. After verifying that the other system components satisfy these assumptions, proof rules are used to establish properties about the combined system. We will first define the basic underlying ideas and then illustrate one of the assume-guarantee proof rules from [69].

The approach uses *probabilistic assume-guarantee triples.* These take the form $\langle\Phi_A\rangle_{\geqslant p_A}\,\mathcal{M}\,\langle\Phi_G\rangle_{\geqslant p_G}$, where $\mathtt{P}_{\geqslant p_A}[\Phi_A]$ and $\mathtt{P}_{\geqslant p_G}[\Phi_G]$ are probabilistic safety properties and $\mathcal{M}$ is a PA. Informally, the triple means: "whenever $\mathcal{M}$ is part of a system satisfying Φ_A with probability at least p_A, the system satisfies Φ_G with probability at least p_G". Formally, we have the following definition.

Definition 30 (Probabilistic assume-guarantee triple). *If* $\mathtt{P}_{\geqslant p_A}[\Phi_A]$ *and* $\mathtt{P}_{\geqslant p_G}[\Phi_G]$ *are probabilistic safety properties,* $\mathcal{M}$ *is a PA and* $\alpha_G \subseteq \alpha_A \cup \alpha_{\mathcal{M}}$, *then* $\langle\Phi_A\rangle_{\geqslant p_A}\,\mathcal{M}\,\langle\Phi_G\rangle_{\geqslant p_G}$ *is a* probabilistic assume-guarantee triple, *meaning:*

$$\forall\sigma{\in}Adv_{\mathcal{M}[\alpha_A]}\ .\ \big(\,Pr^{\sigma}_{\mathcal{M}[\alpha_A],\overline{s}}(\Phi_A){\geqslant}p_A \rightarrow Pr^{\sigma}_{\mathcal{M}[\alpha_A],\overline{s}}(\Phi_G){\geqslant}p_G\,\big)\,.$$

The use of $\mathcal{M}[\alpha_A]$, i.e. $\mathcal{M}$ extended to the alphabet of Φ_A, in the above is needed to allow the assumption to refer to actions not used in $\mathcal{M}$. We use $\langle\mathtt{true}\rangle\,\mathcal{M}\,\langle\Phi_G\rangle_{\geqslant p_G}$ to indicate the case where there is no assumption. This is therefore equivalent to $\overline{s}\models\mathtt{P}_{\geqslant p_G}[\Phi_G]$ and can be verified using the techniques described in Section 7.1. Checking that a triple $\langle\Phi_A\rangle_{\geqslant p_A}\,\mathcal{M}\,\langle\Phi_G\rangle_{\geqslant p_G}$ holds in the general case, however, requires the use of multi-objective (LTL) probabilistic model checking, as discussed in Section 8.

Proposition 4 ([69]). *If* $\mathcal{M}$ *is a PA,* $\mathtt{P}_{\geqslant p_A}[\Phi_A]$ *and* $\mathtt{P}_{\geqslant p_G}[\Phi_G]$ *are probabilistic safety properties and* $\mathcal{M}' = \mathcal{M}[\alpha_A]{\otimes}\mathcal{A}_A^{err}{\otimes}\mathcal{A}_G^{err}$ *with initial state* $\overline{s}'$, *then:*

$$\begin{aligned}&\langle\Phi_A\rangle_{\geqslant p_A}\,\mathcal{M}\,\langle\Phi_G\rangle_{\geqslant p_G}\\ &\iff \neg\exists\sigma'{\in}Adv_{\mathcal{M}'}\ .\ \big(\,Pr^{\sigma'}_{\mathcal{M}',\overline{s}'}(\mathtt{G}\ \neg err_A){\geqslant}p_A \wedge Pr^{\sigma'}_{\mathcal{M}',\overline{s}'}(\mathtt{F}\ err_G){>}1{-}p_G\,\big)\,.\end{aligned}$$

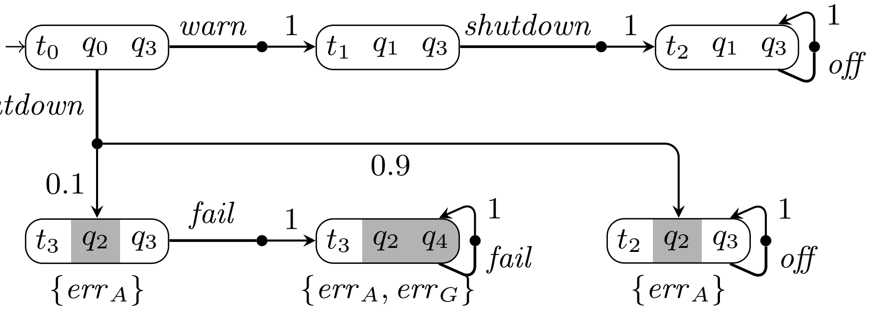


Fig. 17. The product PA $\mathcal{M}_2 \otimes \mathcal{A}^{err}_A \otimes \mathcal{A}^{err}_G$ for the PA $\mathcal{M}_2$ and error automata $\mathcal{A}^{err}_A$ and $\mathcal{A}^{err}_G$ from Figures 8 and 16 (see Example 19)

Based on the definitions given above, [69] presents the following *asymmetric* assume-guarantee proof rule for a two component system $\mathcal{M}_1 \| \mathcal{M}_2$.

Proposition 5 ([69]). *If $\mathcal{M}_1, \mathcal{M}_2$ are PAs and $\mathtt{P}_{\geqslant p_A}[\Phi_A], \mathtt{P}_{\geqslant p_G}[\Phi_G]$ probabilistic safety properties such that $\alpha_A \subseteq \alpha_{\mathcal{M}_1}$ and $\alpha_G \subseteq \alpha_{\mathcal{M}_2} \cup \alpha_A$, then the following proof rule holds:*

$$\frac{\begin{array}{c}\langle \mathtt{true} \rangle\, \mathcal{M}_1\, \langle \Phi_A \rangle_{\geqslant p_A} \\ \langle \Phi_A \rangle_{\geqslant p_A}\, \mathcal{M}_2\, \langle \Phi_G \rangle_{\geqslant p_G}\end{array}}{\langle \mathtt{true} \rangle\, \mathcal{M}_1 \,\|\, \mathcal{M}_2\, \langle \Phi_G \rangle_{\geqslant p_G}} \qquad (\text{ASym})$$

This rule is asymmetric in the sense that it only uses one assumption ($\mathtt{P}_{\geqslant p_A}[\Phi_A]$) about one of the components ($\mathcal{M}_1$). Given such an assumption, we can now model check the probabilistic safety property $\mathtt{P}_{\geqslant p_G}[\Phi_G]$ on $\mathcal{M}_1 \| \mathcal{M}_2$ in a *compositional* fashion. More precisely, verification reduces to two sub-problems, one for each premise of the rule: (i) checking a probabilistic safety property on $\mathcal{M}_1$; (ii) performing multi-objective model checking on $\mathcal{M}_2[\alpha_A] \otimes \mathcal{A}^{err}_A \otimes \mathcal{A}^{err}_G$. If $\mathcal{A}^{err}_A$ is much smaller than $\mathcal{M}_1$, significant gains in performance and/or scalability can be made.

Example 19. We illustrate the rule (ASym) on the PAs $\mathcal{M}_1, \mathcal{M}_2$ and property $\mathtt{P}_{\geqslant 0.98}[\Phi_G]$ from Example 18 (see Figure 16). We also reuse the probabilistic safety property $\mathtt{P}_{\geqslant 0.8}[\Phi_A]$ from Example 13, where Φ_A means "*warn* occurs before *shutdown*" and its error automaton is the DFA $\mathcal{A}^{err}_A$ in Figure 8. Our goal is to verify that $\mathtt{P}_{\geqslant 0.98}[\Phi_G]$ holds in $\mathcal{M}_1 \| \mathcal{M}_2$ using the rule (ASym) and with assumption $\mathtt{P}_{\geqslant 0.8}[\Phi_A]$. To check the first premise of (ASym), we need to verify $s_0 \models \mathtt{P}_{\geqslant 0.8}[\Phi_A]$ in $\mathcal{M}_1$, which has already been shown to be true in Example 13.

Next, we check the second premise, i.e. $\langle \Phi_A \rangle_{\geqslant 0.8}\, \mathcal{M}_2\, \langle \Phi_G \rangle_{\geqslant 0.98}$, using Proposition 4 above. The product $\mathcal{M}_2 \otimes \mathcal{A}^{err}_A \otimes \mathcal{A}^{err}_G$ is shown in Figure 17 (in this case, $\mathcal{M}_2[\alpha_A] = \mathcal{M}_2$). Recall that we label product states corresponding to accepting states of $\mathcal{A}^{err}_A$ and $\mathcal{A}^{err}_G$ with atomic propositions err_A and err_G. We also shade these grey in Figure 17 for clarity. We need to establish that there is *no* adversary under which the probability of remaining within states not satisfying err_A is at least 0.8 *and* the probability of reaching an err_G state is above $1-0.98 = 0.02$. Through a manual inspection, we see that no such adversary exists. This check can be automated using the multi-objective probabilistic model checking techniques from Section 8. In conclusion, combining the two results, we can state that $(s_0, t_0) \models \mathtt{P}_{\geqslant 0.98}[\Phi_G]$ *does* hold in $\mathcal{M}_1 \| \mathcal{M}_1$, as required.

Consider, however, the adversary σ of $\mathcal{M}_2 \otimes \mathcal{A}_A^{err} \otimes \mathcal{A}_G^{err}$ which, in the initial state, chooses *warn* with probability 0.8 and *shutdown* with probability 0.2. This satisfies $\mathtt{G}\ \neg err_A$ with probability 0.8 and $\mathtt{F}\ err_G$ with probability 0.02. Hence, $\langle \Phi_A \rangle_{\geqslant 0.8}\, \mathcal{M}_2\, \langle \Phi_G \rangle_{\geqslant p_G}$ does *not* hold for any value of $p_G > 1-0.02 = 0.98$. ■

For details of additional probabilistic assume-guarantee proof rules, as well as *quantitative* approaches to the problem, see [69]; for further extensions, including the use of ω-regular and reward-based properties, see [45].

10 Tools and Case Studies

There are several software tools available for probabilistic verification of Markov decision processes (or probabilistic automata). One of the most widely used of these is PRISM [56], which incorporates the majority of the techniques described in this tutorial. It also supports discrete- and continuous-time Markov chains, and probabilistic timed automata.

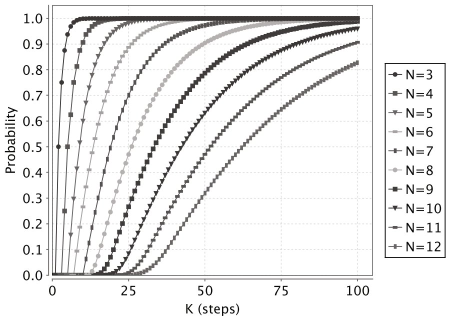
Two other tools for probabilistic model checking of MDPs are LiQuor [30], which has an expressive modelling language extending Promela with probabilities, and ProbDiVinE [13], which focuses on parallel and distributed implementations of LTL model checking for MDPs. RAPTURE [60] and PASS [50] both provide verification of MDPs using abstraction and refinement. There are also various other probabilistic model checkers for discrete- and continuous-time Markov chains, notably MRMC [61]. For a more extensive list, see [92].

In the following sections, we present three large probabilistic model checking case studies, based on the use of Markov decision processes. A selection of further examples can be found at [90].

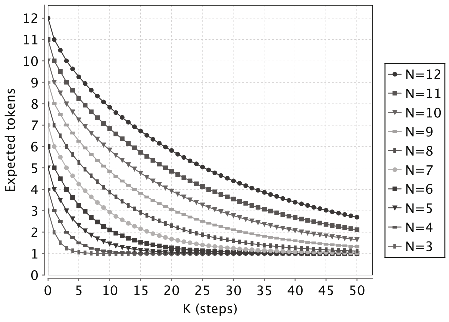
10.1 Case Study: Israeli and Jalfon's Self-stabilisation Protocol

A *self-stabilising protocol* for a network of processes is a protocol which transforms a system from an *unstable* state to a *stable* state in a finite number of steps and without any outside intervention. Here, we consider Israeli and Jalfon's *randomised* self-stabilising protocol [59].

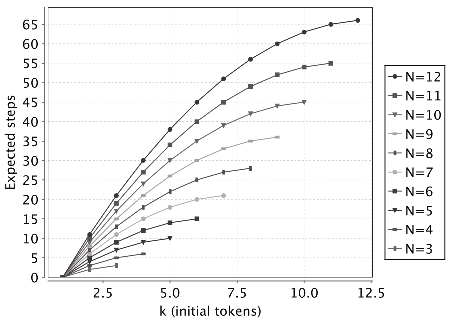
The protocol of Israeli and Jalfon is designed for a network which is an oriented ring of identical processes $P_1, \ldots, P_N$ with bidirectional communication. It operates asynchronously with an arbitrary scheduler, and each process P_i has a boolean variable q_i which represents the fact that it has a token. A process is said to be *active* if it has a token and only active processes can be scheduled. When a process is scheduled, it makes a (uniform) random choice as to whether to move its token to its left or right and when tokens collide they are merged into a single one. The stable configurations are those where there is exactly one active



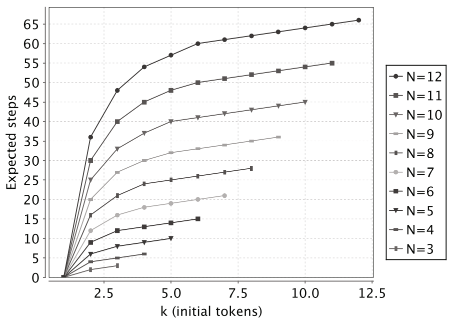
(a) Minimum probability of stabilisation by step K



(b) Maximum expected number of tokens at step K



(c) Minimum expected time to stabilise (initially k tokens)



(d) Maximum expected time to stabilise (initially k tokens)

Fig. 18. Israeli-Jalfon's self-stabilisation protocol: results

process, i.e. exactly one token. Once a stable configuration has been reached, the token should be passed around the ring forever in a fair manner.

We first verify that the protocol does indeed stabilise, by verifying that a stable state is reached with minimum probability 1 for all possible initial configurations. This property can be expressed in PCTL as the formula $\mathtt{P}_{\geqslant 1}[\mathtt{F}\ \mathit{stable}]$ where *stable* is the atomic proposition representing the fact that there is only one token present in the state. We also check that the token is passed around the ring in a fair manner. Since we have already shown that, with minimum probability 1 there is eventually only one token, it is sufficient to check that each process obtains the token infinitely often. For process P_i, we use the probabilistic LTL specification $\mathtt{P}_{\geqslant 1}[\mathtt{G}\ \mathtt{F}\ \mathit{active}_i]$, where the atomic proposition active_i indicates that P_i is active, i.e. has a token.

Next, we investigate the protocol's performance with the properties:

– the minimum probability of stabilising within K steps when starting from any initial configuration, $(\mathtt{P}_{\mathtt{min}=?}[\mathtt{F}^{\leqslant K}\ \mathit{stable}])$;

- the maximum expected number of tokens after K steps when starting from any initial configuration, ($\mathtt{R}^{tokens}_{\mathtt{max}=?}[\mathtt{I}^{=K}]$, where the reward structure *tokens* assigns a reward to each state corresponding to the number of tokens present in the state and there are no action rewards);
- the minimum and maximum expected time to reach a stable state given that the initial number of tokens is k ($\mathtt{R}^{steps}_{\mathtt{min}=?}[\mathtt{F}\ stable]$ and $\mathtt{R}^{steps}_{\mathtt{max}=?}[\mathtt{F}\ stable]$, where the reward structure *steps* assigns a reward of 1 to each action and there are no state rewards).

Figure 18 presents a summary of the results as the number of processes (N) varies from 3 to 12. We see that the performance of the protocol decreases as the number of processes increases. Considering the individual properties, we observe that the probability to stabilise by step K (Figure 18(a)) and the expected number of tokens at step K (Figure 18(b)) both converge towards 1 as K increases. This is to be expected as we have already verified that the minimum probability of stabilisation is 1. Considering the expected time to stabilise (Figures 18(c)-(d)), the results show the expected time to stabilise increasing as the initial number of tokens (k) increases. Further investigation for the other properties shows similar trends as the initial number of tokens is increased.

10.2 Case Study: Dynamic Power Management

Dynamic Power Management (DPM) is a technique for saving energy in devices that can be turned on and off under operating system control. Such methods are particularly important in mobile, hand-held and embedded devices, for which minimisation of energy consumption is a key issue. DPM-enabled devices typically have several *power states* with different energy consumption rates. A DPM *policy* is used to decide when commands to transition between these states should be issued, based on the current state of the system.

The components of a DPM system are: a Service Provider (SP) representing the device under power management control; a Service Requester (SR) which sends requests to the SP; a Service Request Queue (SRQ), which stores the requests that have yet to be served; and the Power Manager (PM), which sends commands to the SP, based on observations of the system and a DPM policy.

The particular system we consider here is the IBM TravelStar VP [58], a commercially available hard disk drive. Our model is based on the one in [15]. The hard disk, i.e. the SP, can operate in five different states as shown in Table 2(a), which also provides the power dissipation in each of these states. It is only in state *active* that the device can perform data reads and writes. In state *idle* the disk is spinning but some of the electronic components of the disk drive have been switched off. The state *idlelp* ("idle low power") is similar except that it has a lower power dissipation. The states *stby* and *sleep* correspond to the disk being spun down. Transition times between states are in Figure 2(b).

We use the following reward structures during our analysis:

- *power* is used to investigate the energy consumption of the system and is defined using the power dissipation of the SP given in Figure 2(a);

Table 2. Dynamic power management: properties of the states of the hard drive

(a) Average power dissipation

State	Power dissipation (W)
active	2.5
idle	1.5
idlelp	0.8
stby	0.3
sleep	0.1

(b) Expected transition times

	active	*idle*	*idlelp*	*stby*	*sleep*
active	-	1ms	5ms	2.2sec	6sec
idle	1ms	-	5ms	2.2sec	6sec
idlelp	5ms	-	-	2.2sec	6sec
stby	2.2sec	-	-	-	6sec
sleep	6sec	-	-	-	-

– *queue* is used for analysing the size of the service request queue and is constructed by setting the reward in each state to the size of the SRQ;
– *lost* represents the loss of requests by assigning a reward of 1 to actions representing the arrival of a request when the queue is full.

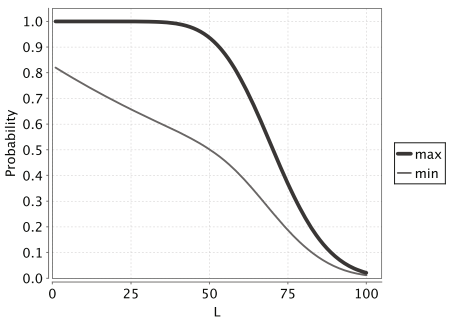
For further details of the PRISM model see [74,91].

In Figure 19 we present model checking results for computation of the minimum and maximum values for the following properties, when the maximum queue size is 2 and there is no constraint on the battery life of the system:

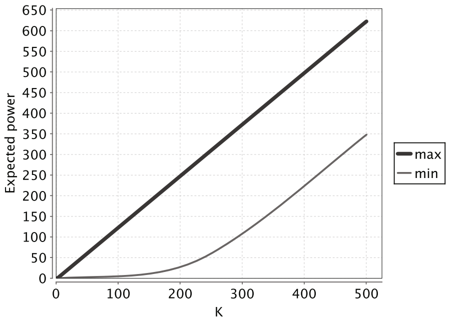
– the probability of L lost requests by step $1,000$ (e.g. $\mathtt{P}_{\mathtt{min}=?}[\mathtt{F}^{\leqslant 1000}\ lost_L]$);
– the expected energy consumed during the first K steps (e.g. $\mathtt{R}^{power}_{\mathtt{min}=?}[\mathtt{C}^{\leqslant K}]$);
– the expected queue size at step K (e.g. $\mathtt{R}^{queue}_{\mathtt{min}=?}[\mathtt{I}^{=K}]$);
– the expected number of lost requests after K steps (e.g. $\mathtt{R}^{lost}_{\mathtt{min}=?}[\mathtt{C}^{\leqslant K}]$).

The results demonstrate that, depending on the power manager's choices, there can be a large difference both in the energy consumption (Figure 19(b)) and in the performance (or quality of service) of the device (Figures 19(a), (c) and (d)). Analysing the best- and worst-case choices, i.e. generating the adversaries that yield the optimal values, we find the best-case performance and worst-case energy consumption is obtained by keeping the SP in the *active* state. On the other hand, the worst-case performance and best-case energy consumption is obtained by keeping the SP in *sleep* whenever possible and only switching to *active* when necessary (to prevent the power manager which minimises energy consumption by ignoring all requests and keeping the SP in sleep, we require the SP to be switched to *active* when the SRQ becomes full).

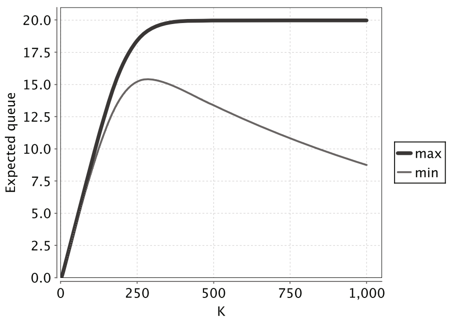
To further investigate the power-versus-performance trade-off of the system, we now apply the techniques of Section 8 and [45] (using the PRISM extension implemented in [69,45]) to perform controller synthesis on the disk-drive. To allow us to consider long-run average properties in the analysis we follow [15] and constrain the battery life of the system. Applying these techniques, we can minimise the expected energy consumption under restrictions on, for example, the probability that a request waits more than K steps, the probability that N requests are lost, the average request-queue size or the expected number of lost requests. To illustrate this approach, Figure 20(a) plots the minimum expected



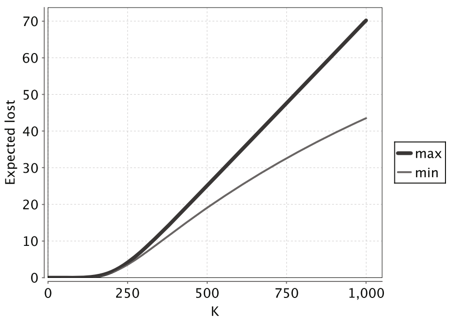
(a) Prob. L lost requests by step 1,000



(b) Expected energy consumed by step K



(c) Expected queue size at step K



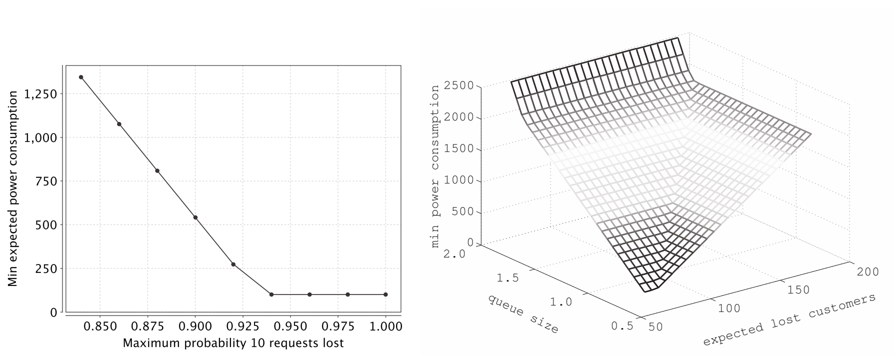
(d) Expected lost requests by step K

Fig. 19. Dynamic power management: model checking results

energy consumption under restrictions on the probability that 10 requests are lost, while Figure 20(b) plots the minimum expected energy consumption under restrictions on both the average queue size and expected number of lost requests. In both cases the maximum size of the SRQ is set to 2. These results demonstrate the familiar power-versus-performance trade-off: policies can offer improved performance, but at the expense of using more energy.

For an example of the controllers generated through this approach, consider constraints of 0.9 and 100 on the average queue size and expected number of lost requests. The optimal expected energy consumption is 1,874.6 and is achieved by the following PM:

– if the SP is in *active*, the SR is in *idle* and the queue is empty, move SP to *idle*;
– if the SP is in *sleep*, the SR is in *idle* and the queue is full, then:
  - with probability 0.972958 keep SP in *sleep*
  - with probability 0.027042 move SP to *active*;
– otherwise, keep the SP in its current state.



(a) Constraint on maximum probability of 10 lost requests (b) Constraints on maximum average queue size and expected lost requests

Fig. 20. Dynamic power management: controller synthesis

On the other hand, if the constraints on the average queue size and the expected lost requests are 0.8 and 80 respectively, then the optimal expected energy consumption is 2,134.5 and is achieved by the following PM:

– immediately move the SP from *sleep* to *active*;
– if the SP is in *idle*, the SR is in *req* and the queue is empty, move SP to *active*;
– if the SP is in *active*, the SR is in *idle* and the queue is non-empty, then
  • with probability 0.995523 move SP to *idle*
  • with probability 0.004477 keep SP in *active*;
– otherwise, keep the SP in its current state.

10.3 Case Study: Aspnes and Herlihy's Consensus Algorithm

A distributed consensus protocol is an algorithm for ensuring that a collection of distributed processes, which start in some initial value supplied by their environment, eventually terminate agreeing on the same value. In this case study, we consider the randomised distributed consensus algorithm of Aspnes & Herlihy [5] and use it to demonstrate the applicability of the compositional verification approach for safety properties introduced in Section 9.

The algorithm of Apnes & Herlihy allows N processes in a distributed network to reach a consensus and employs, in each round, a shared coin protocol parameterised by K. The safety property we will consider is that "agreement is reached by round R", where the corresponding set of bad prefixes are those that end with the action of any process entering round $R{+}1$. We will in fact use the quantitative version of the composition techniques presented in [69], which allows us to compute a lower bound on the minimum probability of satisfying the safety property, by computing an upper bound on the maximum probability of performing a bad prefix (see Proposition 1).

The probabilistic automata model of the algorithm is based on the one presented in [71]. It comprises the parallel composition of: N PAs, each representing

Table 3. Randomised consensus algorithm: performance of compositional verification

Parameters			Non-compositional			Compositional		
N	K	R	Model size	Time (s)	Result†	LP size	Time (s)	Result†
2	2	3	5,158	1.6	0.108333	1,064	**0.9**	0.108333
2	20	3	40,294	108.1	0.012500	1,064	**7.4**	0.012500
2	2	4	20,886	3.6	0.011736	2,372	**1.2**	0.011736
2	20	4	166,614	343.1	0.000156	2,372	**7.8**	0.000156
2	2	5	83,798	7.7	0.001271	4,988	**2.2**	0.001271
2	20	5	671,894	1,347	0.000002	4,988	**8.8**	0.000002
3	2	3	1,418,545	18,971	0.229092	40,542	**29.6**	0.229092
3	12	3	16,674,145*	>24h	-	40,542	**49.7**	0.041643
3	20	3	39,827,233*	>24h	-	40,542	**125.3**	0.024960
3	2	4	150,487,585	78,955	0.052483	141,168	**376.1**	0.052483
3	12	4	1,053,762,385*	mem-out	-	141,168	**396.3**	0.001734
3	20	4	2,028,200,209*	mem-out	-	141,168	**471.9**	0.000623

* These models can be constructed, but not model checked, in PRISM.

† Results are maximum probabilities of error so actual values are these subtracted from 1.

one process, and R PAs, one for the shared coin protocol of each round. The compositional verification consists of the following steps:

- first, using the techniques of Section 7, we calculate the minimum probability that the coin protocols of rounds $1, \ldots, R-2$ each satisfy a safety property (the property is that the coin protocol returns the same coin value for all processes, and therefore the bad prefixes are those where the coin protocol returns different values to different processes);
- second, we combine these results through $R-2$ applications of of the Async rule of [69] to return a probabilistic safety property satisfied by the (asynchronous) composition of the shared coin protocols for the first $R-2$ rounds;
- finally, this probabilistic safety property is used as the assumption for an application of the Asym rule (see Theorem 5), yielding the final property of interest on the combined system: the minimum probability that agreement is reached by round R.

Table 3 shows performance results (taken from [69]) for compositional verification on this case study. It gives the total time required to perform verification, both compositionally (as above) and non-compositionally (using PRISM). To give an indication of the improvements in scalability, the table shows the size of the PA (number of states) for the full system and the number of variables in the LP problems used for multi-objective model checking in the compositional case (see Section 8). As can be seen, for this case study, the compositional approach is faster and permits analysis of models that are infeasible with conventional (non-compositional) techniques. See [69] for further details.

11 Conclusions and Further Reading

This tutorial has given a general introduction to probabilistic model checking, focusing on the model of Markov decision processes. We have covered the basic underlying theory for this model, discussed techniques for model checking a wide array of quantitative properties and presented some illustrative case studies.

We conclude by briefly outlining a few of the active research topics in this area and give some pointers to further reading. In the context of automated verification of probabilistic systems, several key challenges are being addressed. One is extending the range of models to which probabilistic model checking can be applied. Another is improving the scalability of the techniques to handle larger and more complex models. There are also many other ways in which the functionality and applicability of probabilistic verification are being improved.

Models. Recent advances have been made regarding model checking for the following extensions of MDPs: probabilistic timed automata (see e.g. [70] for a survey); probabilistic hybrid systems (see e.g. [88,46]); continuous-time MDPs and continuous-time games (see e.g. [10,23,73,77]); interactive Markov chains (see e.g. [87]); and recursive MDPs and games (see e.g. [41,22]).

Scalability. A variety of approaches are being considered to improve scalability. One example is the development of *abstraction and refinement* frameworks [37,55,26,63], some of which have been applied in practice to verification of probabilistic timed automata [68], probabilistic software [62] and PRISM models [50]. Other promising directions include: *partial order reduction* [49,31], *symmetry reduction* [66,39], algorithms for *simulation* and *bisimulation* relations [25,86] and *compositional* probabilistic verification techniques [69,43,38].

Other directions. Many other interesting topics are being studied on MDPs and related models. These include: *probabilistic counterexample* generation [4,3], verification under *fairness* [8] and under *restricted classes of adversaries* [47,29], *parametric* model checking [51], *synthesis* of parameters [52] and models [28], and *run-time* probabilistic model checking [24,44].

Acknowledgments

The authors are supported in part by EU FP7 project CONNECT, ERC Advanced Grant VERIWARE and EPSRC grant EP/F001096/1. Vojtěch Forejt is also supported by a Royal Society Newton Fellowship and the Institute for Theoretical Computer Science, project no. 1M0545. In addition, we acknowledge Dinara Karkabayeva for providing a preliminary version of the results in Section 4.2 and thank the authors of [15] for making their model of the IBM TravelStar VP available to us.

References

1. de Alfaro, L.: Formal Verification of Probabilistic Systems. Ph.D. thesis, Stanford University (1997)
2. de Alfaro, L.: From fairness to chance. In: Baier, C., Huth, M., Kwiatkowska, M., Ryan, M. (eds.) Proc. 1st Int. Workshop Probabilistic Methods in Verification (PROBMIV 1998). ENTCS, vol. 22. Elsevier, Amsterdam (1998)
3. Aljazzar, H., Leue, S.: Generation of counterexamples for model checking of Markov decision processes. In: Proc. 6th Int. Conf. Quantitative Evaluation of Systems (QEST 2009), pp. 197–206. IEEE CS Press, Los Alamitos (2009)
4. Andrés, M., D'Argenio, P., van Rossum, P.: Significant diagnostic counterexamples in probabilistic model checking. In: Chockler, H., Hu, A. (eds.) HVC 2008. LNCS, vol. 5394, pp. 129–148. Springer, Heidelberg (2009)
5. Aspnes, J., Herlihy, M.: Fast randomized consensus using shared memory. Journal of Algorithms 15(1), 441–460 (1990)
6. Aziz, A., Singhal, V., Balarin, F., Brayton, R., Sangiovanni-Vincentelli, A.: It usually works: The temporal logic of stochastic systems. In: Wolper, P. (ed.) CAV 1995. LNCS, vol. 939, pp. 155–165. Springer, Heidelberg (1995)
7. Baier, C.: On algorithmic verification methods for probabilistic systems, habilitation thesis, Fakultät für Mathematik & Informatik, Universität Mannheim (1998)
8. Baier, C., Groesser, M., Ciesinski, F.: Quantitative analysis under fairness constraints. In: Liu, Z., Ravn, A. (eds.) ATVA 2009. LNCS, vol. 5799, pp. 135–150. Springer, Heidelberg (2009)
9. Baier, C., Größer, M., Leucker, M., Bollig, B., Ciesinski, F.: Controller synthesis for probabilistic systems. In: Lévy, J.J., Mayr, E., Mitchell, J. (eds.) Proc. 3rd IFIP Int. Conf. Theoretical Computer Science (TCS 2006), pp. 493–506. Kluwer, Dordrecht (2004)
10. Baier, C., Hermanns, H., Katoen, J.P., Haverkort, B.: Efficient computation of time-bounded reachability probabilities in uniform continuous-time Markov decision processes. Theoretical Computer Science 345(1), 2–26 (2005)
11. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press, Cambridge (2008)
12. Baier, C., Kwiatkowska, M.: Model checking for a probabilistic branching time logic with fairness. Distributed Computing 11(3), 125–155 (1998)
13. Barnat, J., Brim, L., Cerna, I., Ceska, M., Tumova, J.: ProbDiVinE-MC: Multi-core LTL model checker for probabilistic systems. In: Proc. 5rd Int. Conf. Quantitative Evaluation of Systems (QEST 2008), pp. 77–78. IEEE CS Press, Los Alamitos (2008)
14. Bellman, R.: Dynamic Programming. Princeton University Press, Princeton (1957)
15. Benini, L., Bogliolo, A., Paleologo, G., De Micheli, G.: Policy optimization for dynamic power management. IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems 8(3), 299–316 (2000)
16. Bertsekas, D.: Dynamic Programming and Optimal Control, vol. 1,2. Athena Scientific, Belmont (1995)
17. Bertsekas, D., Tsitsiklis, J.: An analysis of stochastic shortest path problems. Mathematics of Operations Research 16(3), 580–595 (1991)
18. Bianco, A., de Alfaro, L.: Model checking of probabilistic and nondeterministic systems. In: Thiagarajan, P. (ed.) FSTTCS 1995. LNCS, vol. 1026, pp. 499–513. Springer, Heidelberg (1995)

19. Billingsley, P.: Probability and Measure. Wiley, Chichester (1995)
20. Brázdil, T., Forejt, V., Kučera, A.: Controller synthesis and verification for Markov decision processes with qualitative branching time objectives. In: Aceto, L., Damgård, I., Goldberg, L., Halldórsson, M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008, Part II. LNCS, vol. 5126, pp. 148–159. Springer, Heidelberg (2008)
21. Brázdil, T., Brožek, V., Forejt, V., Kučera, A.: Stochastic games with branching-time winning objectives. In: 21th IEEE Symp. Logic in Computer Science (LICS 2006), pp. 349–358. IEEE CS Press, Los Alamitos (2006)
22. Brázdil, T., Brožek, V., Kučera, A., Obdržálek, J.: Qualitative reachability in stochastic BPA games. In: Albers, S., Marion, J.Y. (eds.) 26th Int. Symp. Theoretical Aspects of Computer Science (STACS 2009). LIPIcs, vol. 3, pp. 207–218. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2009)
23. Brázdil, T., Forejt, V., Krčál, J., Křetínský, J., Kučera, A.: Continuous-time stochastic games with time-bounded reachability. In: Kannan, R., Kumar, K. (eds.) Proc. 29th Int. Conf. Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2009). LIPIcs, vol. 4, pp. 61–72. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2009)
24. Calinescu, R., Grunske, L., Kwiatkowska, M., Mirandola, R., Tamburrelli, G.: Dynamic QoS management and optimisation in service-based systems. IEEE Transactions on Software Engineering (2010)
25. Cattani, S., Segala, R.: Decision algorithms for probabilistic bisimulation. In: Brim, L., Janar, P., Ketinsky, M., Kuera, A. (eds.) CONCUR 2002. LNCS, vol. 2421, pp. 371–385. Springer, Heidelberg (2002)
26. Chadha, R., Viswanathan, M.: A counterexample guided abstraction-refinement framework for Markov decision processes. ACM Transactions on Computational Logic 12(1), 1–49 (2010)
27. Chatterjee, K., Henzinger, T.: Value iteration. In: Grumberg, O., Veith, H. (eds.) 25 Years of Model Checking. LNCS, vol. 5000, pp. 107–138. Springer, Heidelberg (2008)
28. Chatterjee, K., Henzinger, T., Jobstmann, B., Singh, R.: Measuring and synthesizing systems in probabilistic environments. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 380–395. Springer, Heidelberg (2010)
29. Cheung, L.: Reconciling Nondeterministic and Probabilistic Choices. Ph.D. thesis, Radboud University of Nijmegen (2006)
30. Ciesinski, F., Baier, C.: Liquor: A tool for qualitative and quantitative linear time analysis of reactive systems. In: Proc. 3rd Int. Conf. Quantitative Evaluation of Systems (QEST 2006), pp. 131–132. IEEE CS Press, Los Alamitos (2006)
31. Ciesinski, F., Baier, C., Größer, M., Parker, D.: Reduction techniques for model checking Markov decision processes. In: Proc. 5th Int. Conf. Quantitative Evaluation of Systems (QEST 2008), pp. 45–54. IEEE CS Press, Los Alamitos (2008)
32. Ciesinski, F., Größer, M.: On probabilistic computation tree logic. In: Baier, C., Haverkort, B., Hermanns, H., Katoen, J.P., Siegle, M. (eds.) Validation of Stochastic Systems. LNCS, vol. 2925, pp. 147–188. Springer, Heidelberg (2004)
33. Clarke, E., Emerson, A.: Design and synthesis of synchronization skeletons using branching time temporal logic. In: Kozen, D. (ed.) Logic of Programs 1981. LNCS, vol. 131, pp. 52–71. Springer, Heidelberg (1982)

34. Courcoubetis, C., Yannakakis, M.: Verifying temporal properties of finite state probabilistic programs. In: Proc. 29th Annual Symp. Foundations of Computer Science (FOCS 1988), pp. 338–345. IEEE CS Press, Los Alamitos (1988)
35. Courcoubetis, C., Yannakakis, M.: Markov decision processes and regular events. IEEE Trans. Automatic Control 43(10), 1399–1418 (1998)
36. Daniele, M., Giunchiglia, F., Vardi, M.: Improved automata generation for linear temporal logic. In: Halbwachs, N., Peled, D. (eds.) CAV 1999. LNCS, vol. 1633, pp. 249–260. Springer, Heidelberg (1999)
37. D'Argenio, P., Jeannet, B., Jensen, H., Larsen, K.: Reduction and refinement strategies for probabilistic analysis. In: Hermanns, H., Segala, R. (eds.) PROBMIV 2002, PAPM-PROBMIV 2002, and PAPM 2002. LNCS, vol. 2399, pp. 57–76. Springer, Heidelberg (2002)
38. Delahaye, B., Caillaud, B., Legay, A.: Probabilistic contracts: A compositional reasoning methodology for the design of stochastic systems. In: Proc. 10th Int. Conf. Application of Concurrency to System Design (ACSD 2010), pp. 223–232. IEEE CS Press, Los Alamitos (2010)
39. Donaldson, A., Miller, A.: Symmetry reduction for probabilistic model checking using generic representatives. In: Graf, S., Zhang, W. (eds.) ATVA 2006. LNCS, vol. 4218, pp. 9–23. Springer, Heidelberg (2006)
40. Etessami, K., Kwiatkowska, M., Vardi, M., Yannakakis, M.: Multi-objective model checking of Markov decision processes. Logical Methods in Computer Science 4(4), 1–21 (2008)
41. Etessami, K., Yannakakis, M.: Recursive Markov decision processes and recursive stochastic games. In: Caires, L., Italiano, G., Monteiro, L., Palamidessi, C., Yung, M. (eds.) ICALP 2005. LNCS, vol. 3580, pp. 891–903. Springer, Heidelberg (2005)
42. Feller, W.: An Introduction to Probability Theory and its Applications. Wiley, Chichester (1968)
43. Feng, L., Kwiatkowska, M., Parker, D.: Compositional verification of probabilistic systems using learning. In: Proc. 7th Int. Conf. Quantitative Evaluation of Systems (QEST 2010), pp. 133–142. IEEE CS Press, Los Alamitos (2010)
44. Filieri, A., Ghezzi, C., Tamburrelli, G.: Run-time efficient probabilistic model checking. In: Proc. 33rd ACM/IEEE International Conference on Software Engineering (ICSE 2011). ACM, New York (2011)
45. Forejt, V., Kwiatkowska, M., Norman, G., Parker, D., Qu, H.: Quantitative multi-objective verification for probabilistic systems. In: Abdulla, P., Leino, K. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 112–127. Springer, Heidelberg (2011)
46. Fränzle, M., Teige, T., Eggers, A.: Engineering constraint solvers for automatic analysis of probabilistic hybrid automata. Journal of Logic and Algebraic Programming 79(7), 436–466 (2010)
47. Giro, S.: On the automatic verification of distributed probabilistic automata with partial information. Ph.D. thesis, FaMAF, Universidad Nacional de Córdoba (2010)
48. van Glabbeek, R., Smolka, S., Steffen, B.: Reactive, generative, and stratified models of probabilistic processes. Information and Computation 121(1), 59–80 (1995)
49. Größer, M., Baier, C.: Partial order reduction for Markov decision processes: A survey. In: de Boer, F., Bonsangue, M., Graf, S., de Roever, W.P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 408–427. Springer, Heidelberg (2006)
50. Hahn, E., Hermanns, H., Wachter, B., Zhang, L.: PASS: Abstraction refinement for infinite probabilistic models. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 353–357. Springer, Heidelberg (2010)

51. Hahn, E., Hermanns, H., Zhang, L.: Probabilistic reachability for parametric Markov models. In: Pasareanu, C. (ed.) Model Checking Software. LNCS, vol. 5578, pp. 88–106. Springer, Heidelberg (2009)
52. Han, T., Katoen, J.P., Mereacre, A.: Approximate parameter synthesis for probabilistic time-bounded reachability. In: Proc. IEEE Symp. Real-Time Systems (RTSS 2008), pp. 173–182. IEEE CS Press, Los Alamitos (2008)
53. Hansson, H.: Time and Probability in Formal Design of Distributed Systems. Elsevier, Amsterdam (1994)
54. Hansson, H., Jonsson, B.: A logic for reasoning about time and reliability. Formal Aspects of Computing 6(5), 512–535 (1994)
55. Hermanns, H., Wachter, B., Zhang, L.: Probabilistic CEGAR. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 162–175. Springer, Heidelberg (2008)
56. Hinton, A., Kwiatkowska, M., Norman, G., Parker, D.: PRISM: A tool for automatic verification of probabilistic systems. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 441–444. Springer, Heidelberg (2006)
57. Howard, R.: Dynamic Programming and Markov Processes. MIT Press, Cambridge (1960)
58. Technical specifications of hard drive IBM Travelstar VP, `http://www.storage.ibm.com/storage/oem/data/travvp.htm`
59. Israeli, A., Jalfon, M.: Token management schemes and random walks yield self-stabilizating mutual exclusion. In: Proc. 9th Annual ACM Symp. Principles of Distributed Computing (PODC 1990), pp. 119–131. ACM, New York (1990)
60. Jeannet, B., D'Argenio, P., Larsen, K.: Rapture: A tool for verifying Markov decision processes. In: Cerna, I. (ed.) Proc. Tools Day, affiliated to 13th Int. Conf. Concurrency Theory (CONCUR 2002), pp. 84–98 (2002); Technical Report FIMU-RS-2002-05, Faculty of Informatics, Masaryk University (2002)
61. Katoen, J.P., Hahn, E., Hermanns, H., Jansen, D., Zapreev, I.: The ins and outs of the probabilistic model checker MRMC. In: Proc. 6th Int. Conf. Quantitative Evaluation of Systems (QEST 2009), pp. 167–176. IEEE CS Press, Los Alamitos (2009)
62. Kattenbelt, M., Kwiatkowska, M., Norman, G., Parker, D.: Abstraction refinement for probabilistic software. In: Jones, N., Muller-Olm, M. (eds.) VMCAI 2009. LNCS, vol. 5403, pp. 182–197. Springer, Heidelberg (2009)
63. Kattenbelt, M., Kwiatkowska, M., Norman, G., Parker, D.: A game-based abstraction-refinement framework for Markov decision processes. Formal Methods in System Design 36(3) (2010)
64. Kemeny, J., Snell, J., Knapp, A.: Denumerable Markov Chains, 2nd edn. Springer, Heidelberg (1976)
65. Kulkarni, V.: Modeling and Analysis of Stochastic Systems. Chapman and Hall, Boca Raton (1995)
66. Kwiatkowska, M., Norman, G., Parker, D.: Symmetry reduction for probabilistic model checking. In: Ball, T., Jones, R. (eds.) CAV 2006. LNCS, vol. 4144, pp. 234–248. Springer, Heidelberg (2006)
67. Kwiatkowska, M., Norman, G., Parker, D.: Stochastic model checking. In: Bernardo, M., Hillston, J. (eds.) SFM 2007. LNCS, vol. 4486, pp. 220–270. Springer, Heidelberg (2007)
68. Kwiatkowska, M., Norman, G., Parker, D.: Stochastic games for verification of probabilistic timed automata. In: Ouaknine, J., Vaandrager, F. (eds.) FORMATS 2009. LNCS, vol. 5813, pp. 212–227. Springer, Heidelberg (2009)

69. Kwiatkowska, M., Norman, G., Parker, D., Qu, H.: Assume-guarantee verification for probabilistic systems. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 23–37. Springer, Heidelberg (2010)
70. Kwiatkowska, M., Norman, G., Parker, D., Sproston, J.: Verification of Real-Time Probabilistic Systems. In: Modeling and Verification of Real-Time Systems: Formalisms and Software Tools, pp. 249–288. John Wiley & Sons, Chichester (2008)
71. Kwiatkowska, M., Norman, G., Segala, R.: Automated verification of a randomized distributed consensus protocol using cadence SMV and PRISM. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 194–206. Springer, Heidelberg (2001)
72. Legay, A., Murawski, A., Ouaknine, J., Worrell, J.: On automated verification of probabilistic programs. In: Ramakrishnan, C., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 173–187. Springer, Heidelberg (2008)
73. Neuhäußer, M., Zhang, L.: Time-bounded reachability probabilities in continuous-time Markov decision processes. In: Proc. 7th Int. Conf. Quantitative Evaluation of Systems (QEST 2010), pp. 209–218. IEEE CS Press, Los Alamitos (2010)
74. Norman, G., Parker, D., Kwiatkowska, M., Shukla, S., Gupta, R.: Using probabilistic model checking for dynamic power management. Formal Aspects of Computing 17(2), 160–176 (2005)
75. Pnueli, A.: The temporal logic of programs. In: Proc. 18th Annual Symp. Foundations of Computer Science (FOCS 1977), pp. 46–57. IEEE CS Press, Los Alamitos (1977)
76. Puterman, M.: Markov Decision Processes: Discrete Stochastic Dynamic Programming. John Wiley and Sons, Chichester (1994)
77. Rabe, M., Schewe, S.: Optimal time-abstract schedulers for CTMDPs and Markov games. In: Di Pierro, A., Norman, G. (eds.) Proc. 8th Workshop Quantitative Aspects of Programming Languages (QAPL 2010). EPTCS, vol. 28, pp. 144–158. Open Publishing Association (2010)
78. Rabin, M.: Probabilistic automata. Information and Control 6, 230–245 (1963)
79. Roscoe, A.: The theory and practice of concurrency. Prentice-Hall, Englewood Cliffs (1997)
80. Segala, R.: Modelling and Verification of Randomized Distributed Real Time Systems. Ph.D. thesis, Massachusetts Institute of Technology (1995)
81. Segala, R., Lynch, N.: Probabilistic simulations for probabilistic processes. Nordic Journal of Computing 2(2), 250–273 (1995)
82. Sokolova, A., de Vink, E.: Probabilistic automata: System types, parallel composition and comparison. In: Baier, C., Haverkort, B., Hermanns, H., Katoen, J.P., Siegle, M. (eds.) Validation of Stochastic Systems. LNCS, vol. 2925, pp. 1–43. Springer, Heidelberg (2004)
83. Stewart, W.: Introduction to the Numerical Solution of Markov Chains, Princeton (1994)
84. Vardi, M.: Automatic verification of probabilistic concurrent finite state programs. In: Proc. 26th Annual Symp. Foundations of Computer Science (FOCS 1985), pp. 327–338. IEEE CS Press, Los Alamitos (1985)
85. Vardi, M., Wolper, P.: Reasoning about infinite computations. Information and Computation 115(1), 1–37 (1994)

86. Zhang, L., Hermanns, H.: Deciding simulations on probabilistic automata. In: Namjoshi, K., Yoneda, T., Higashino, T., Okamura, Y. (eds.) ATVA 2007. LNCS, vol. 4762, pp. 207–222. Springer, Heidelberg (2007)
87. Zhang, L., Neuhäußer, M.: Model checking interactive Markov chains. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 53–68. Springer, Heidelberg (2010)
88. Zhang, L., She, Z., Ratschan, S., Hermanns, H., Hahn, E.: Safety verification for probabilistic hybrid systems. In: Cook, B., Jackson, P., Touili, T. (eds.) CAV 2010. LNCS, vol. 6174, pp. 196–211. Springer, Heidelberg (2010)
89. `http://www.prismmodelchecker.org/benchmarks/`
90. `http://www.prismmodelchecker.org/casestudies/`
91. `http://www.prismmodelchecker.org/files/sfm11/`
92. `http://www.prismmodelchecker.org/other-tools.php`

Modeling and Verification of Components and Connectors

Christel Baier, Joachim Klein, and Sascha Klüppelholz

Faculty of Computer Science,
Technische Universität Dresden, Germany

Abstract. Component-based software engineering divides a complex system into smaller logical components with well-defined interfaces. To likewise make the complex interactions between components explicit, exogenous coordination languages like Reo allow the construction of complex coordination glue code in the form of networks of channels and connectors, orchestrating the interactions of the components. In this paper, we present an overview of the modeling concepts for components and connectors using Reo and the underlying constraint automata framework and detail the specification and verification of properties using logics tailored to this framework.

1 Introduction

The main idea of component-based software engineering is to divide a complex system into smaller logical components with well-defined interfaces. For this purpose, a variety of formalisms [23,28,24] have been introduced to capture the behavior of the components as well as the interaction between the components in a manner allowing the application of model checking techniques [17,9].

In this article, we explore the modeling and formal verification of models specified in the Reo [2] and constraint automaton [10] framework. Reo is a channel-based, exogenous coordination language, where the glue code that organizes the interactions of the components is provided by a network of channels. In the exogenous setting, the components themselves are not aware of the context in which they are used, providing a clean separation between computation inside the components and coordination. To facilitate hierarchical modeling, a Reo network can be regarded as a component connector and can then be used in a higher level network of the model as a basic building block, hiding the internal behavior and implementation details. A library of commonly used channels capturing various synchronous and asynchronous behavior and connectors implementing common coordination patterns, as well as the ability to use custom channels and component connectors allows for the modeling of a wide variety of communication and coordination scenarios. Constraint automata serve as the uniform operational semantics for both the interface behavior of the components as well as for the coordination mechanisms arising from the connector glue. The constraint automata semantics of Reo is compositional, i.e., the behavior of the constituent parts of a Reo network, i.e., channels, connectors and the nodes where channels are joined together, can be captured by constraint automata and a product automata construction yields the composite behavior.

Adapting well-known formal verification techniques such as model checking of linear-time [29,31] or branching-time [16] properties to the Reo and constraint automata framework thus allows the specification and verification of coordination mechanisms and components, both in isolation and in concert.

Outline. We first present the basic principles of constraint automata in Section 2 and provide an overview of the coordination language Reo and its constraint automata semantics in Section 3. In Section 4 we present the logics $\mathtt{LTL}_{\mathrm{IO}}$ [8] and $\mathtt{BTSL}$ [26] for specifying linear-time and branching-time properties in the Reo and constraint automata context. We also provide a brief description of the model checking algorithms used to verify such properties and for checking bisimulation equivalence [8,11]. Furthermore, we sketch the main features of our modeling and verification tool-kit Vereofy [12,7]. Section 5 addresses the synthesis problem where a component connector is given as a constraint automaton $\mathcal{A}$ and where the task is to provide Reo code (a network of channels) realizing $\mathcal{A}$.

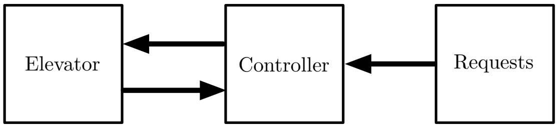


Fig. 1. A simple model for an elevator control system

As a running example in this article, we will use a simple elevator control system. An overview of the basic structure of the system is depicted in Fig. 1. The *Elevator* component models the elevator, keeping track of the current level of the elevator and responding to action requests, i.e., to move up or down. The *Requests* component models the generation of requests by the users, while the *Controller* is then responsible for controlling the elevator. The controller can query the elevator for its current location and send action requests, as well as accept new user requests.

2 Constraint Automata

Constraint automata (CA) [10] provide a generic operational model to formalize the behavioral interfaces of the components, the network that coordinates the components (i.e., the glue code or connector), and the composite system consisting of the components and the glue code. Constraint automata are variants of labeled transition systems (LTS) where the labels of the transitions represent the (possibly data-dependent) I/O-operations of the components and the network. They support any kind of synchronous and asynchronous peer-to-peer communication. The states of a constraint automaton represent the local states of components and/or configurations of a connector.

To formalize the I/O-activity, constraint automata use a finite set $\mathcal{N}$ of data-flow locations, where each element of $A \in \mathcal{N}$ stands for a data-flow location where I/O can occur, such as the interface ports of components, nodes in the connector network, etc. To simplify the presentation in this paper, we assume that the data items that may occur at each data-flow location are elements of a finite, global data domain Data. Each transition of a constraint automaton is labeled by a pair (N, g), where $N \subseteq \mathcal{N}$ is a set of active data-flow locations and g is a data constraint which restricts the possible data items at the active data-flow locations in N. Formally, data constraints are propositional formulas built from the atoms "$d_A = d$", where data item $d \in \mathsf{Data}$ occurs at data-flow location $A \in \mathcal{N}$, and "$d_A = d_B$", where the data items at data-flow locations A and B, with $A, B \in \mathcal{N}$, are the same.

Definition 1 (Data constraints). Data constraints are given by the following grammar:

$$g \quad ::= \quad \text{true} \quad | \quad d_A = d \quad | \quad d_A = d_B \quad | \quad g_1 \vee g_2 \quad | \quad \neg g$$

where $A, B \in \mathcal{N}$ and $d \in \mathsf{Data}$. For a subset $N \subseteq \mathcal{N}$, we denote the set of data constraints using only atoms of the form "$d_A = d$" and "$d_A = d_B$" with $A, B \in N$ by $DC(N)$. Other standard propositional operators such as conjunction ($\wedge$) or implication ($\rightarrow$) can be derived as usual. $d_A \neq d$ stands shortly for $\neg(d_A = d)$. □

Definition 2 (Constraint automata). A constraint automaton is a tuple $\mathcal{A} = (Q, \mathcal{N}, \mathcal{N}_{\text{in}}, \mathcal{N}_{\text{out}}, \longrightarrow, Q_0)$ where

- Q is a finite set of states,
- $\mathcal{N}$ is a finite set of data-flow locations,
- $\mathcal{N}_{\text{in}}$ and $\mathcal{N}_{\text{out}}$ are disjoint subsets of $\mathcal{N}$,
- $\longrightarrow$ is a subset of $Q \times 2^{\mathcal{N}} \times DC(\mathcal{N}) \times Q$,
- $Q_0 \subseteq Q$ is the non-empty set of initial states.

We write $q \xrightarrow{N,g} p$ instead of $(q, N, g, p) \in \longrightarrow$. For every transition $q \xrightarrow{N,g} p$, we require that $g \in DC(N)$, i.e., that the data constraint only refers to data at the active data-flow locations $A \in N$. □

To simplify the presentation of this article, we describe our logical approach under the assumption that there are no terminating behaviors. This assumption can be seen as requiring that no state in the constraint automaton is terminal, i.e., for every state $q \in Q$ there is at least one outgoing transition $q \xrightarrow{N,g} p$ with g being a satisfiable data constraint. This assumption is somehow unrealistic, as deadlock situations may arise in practice, e.g., when the requested interactions of the components are contradicting. The simultaneous treatment of infinite and terminating behavior causes some minor technical difficulties (e.g. [26]), which are avoided here for the sake of a clear and simple presentation of the major concepts.

The subsets $\mathcal{N}_{\text{in}}$ and $\mathcal{N}_{\text{out}}$ of $\mathcal{N}$ provide a characterization of the corresponding data-flow locations as being available for an external connection, which later becomes important during the composition of the constraint automata for the components and the network. Data-flow locations in $\mathcal{N}$ that are not elements of either $\mathcal{N}_{\text{in}}$ or $\mathcal{N}_{\text{out}}$ can be regarded as internal data-flow locations.

Due to the data constraint, each transition stands for a *set of concurrent I/O-operations*, which formalizes the assignment of concrete data values to the active data-flow locations.

Definition 3 (Concurrent I/O-operations (CIO)).
A concurrent I/O-operation is a partial function assigning data values to the data-flow locations, i.e., a function $c : \mathcal{N} \rightarrow \mathsf{Data} \cup \{\bot\}$, where the symbol $\bot$ means "undefined". We write *active*(c) for the set of data-flow locations $A \in \mathcal{N}$ with $c(A) \in \mathsf{Data}$. The *empty* concurrent I/O-operation, denoted ε, is the unique concurrent I/O-operation where *active*$(\varepsilon) = \varnothing$. $\mathsf{CIO}_{\mathcal{N}}$, or briefly CIO, denotes the set of all concurrent I/O-operations (including ε). The set of concurrent I/O-operations consistent with a transition label N, g is then defined as:

$$\mathsf{CIO}(N,g) \;\stackrel{\text{def}}{=}\; \{c \in \mathsf{CIO} : \mathit{active}(c) = N \;\wedge\; c \models g\},$$

where $c \models g$ stands for the obvious satisfaction relation which results from interpreting the data constraint g over the data assignments given by c. □

Remark 1. The empty concurrent I/O-operation ε represents any step in the automaton where no data flow at some $A \in \mathcal{N}$ is observable. It can represent an internal step of some component or a non-observable step of the coordination network where data flow appears at most at some "hidden nodes" of the network (cf. Def. 6 in Sec. 3).

Definition 4 (Executions, paths, I/O-streams)
Let $\mathcal{A} = (Q, \mathcal{N}, \mathcal{N}_{\text{in}}, \mathcal{N}_{\text{out}}, \longrightarrow, Q_0)$ be a constraint automaton. An *execution* in $\mathcal{A}$ is a finite or infinite sequence

$$\eta \;=\; q_0 \xrightarrow{c_1} q_1 \xrightarrow{c_2} \ldots$$

where, for all $i \geq 0$, $q_i \in Q$, $c_i \in \mathsf{CIO}$, and $c_i \in \mathsf{CIO}(N,g)$ for some transition $q_i \xrightarrow{N,g} q_{i+1}$ in $\mathcal{A}$, i.e., where each step from state to state is a valid concurrent I/O-operation in the automaton.

As stated above, in this paper we focus on infinite behaviors and assume that for each state $q \in Q$ there is at least one transition $q \xrightarrow{N,g} p$ such that $g \not\equiv$ false. We define a *path* of $\mathcal{A}$ to be an infinite execution and write $\mathsf{Paths}(q)$ to denote the set of all paths starting in state $q \in Q$. Let $\pi = q_0 \xrightarrow{c_1} q_1 \xrightarrow{c_2} \ldots$ be a path and $0 \leq n$. Then, $\pi\!\downarrow\! n$ denotes the prefix of path π with length n and $\pi\!\uparrow\! n$ the suffix starting at the n-th state. Thus,

$$\begin{aligned}
\pi\!\downarrow\! n &\stackrel{\text{def}}{=} q_0 \xrightarrow{\;c_1\;} \ldots \xrightarrow{\;c_n\;} q_n \\
\pi\!\uparrow\! n &\stackrel{\text{def}}{=} q_n \xrightarrow{c_{n+1}} q_{n+1} \xrightarrow{c_{n+2}} q_{n+2} \xrightarrow{c_{n+3}} \ldots .
\end{aligned}$$

The notion of an *I/O-stream* for constraint automata corresponds to action sequences in LTS. The I/O-stream $\mathsf{ios}(\eta)$ of a finite execution η is the finite word over CIO that is obtained by taking the projection to the labels of the transitions. Formally, if $\eta = q_0 \xrightarrow{c_1} \dots \xrightarrow{c_n} q_n$ is a finite execution then $\mathsf{ios}(\eta) \overset{\text{def}}{=} c_1 \dots c_n \in \mathsf{CIO}^*$. □

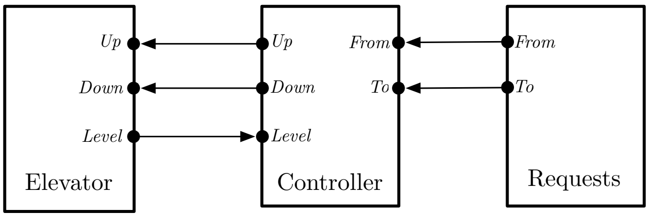


Fig. 2. A refined version of the elevator control system

Example 1. Fig. 2 shows a refined version of the elevator control system as presented earlier in the introduction of this paper. Here we made the communication structure more explicit. The *Elevator* component may receive action requests via its interface ports *Up* and *Down*, resulting in a move of the elevator by one level. The current position of the elevator can be queried using its interface port *Level*.

The *Requests* component modeling the user requests has two interface ports, which provide the current level of the user requesting the elevator (*From*) and the destination level (*To*). We assume here that the users provide their desired destination at the moment they request the elevator. The *Controller* component has corresponding interface ports for communicating with the *Elevator* and *Requests* component.

We now consider constraint automata for parts of the elevator system. For an elevator system with k levels, we assume the global data domain $\mathsf{Data} = \{1, \dots, k\}$, i.e., a finite set of natural numbers for encoding the different requests and the level information.

Fig. 3 shows a constraint automaton for the *Elevator* component for three levels. As the data values for the action requests *Up* and *Down* are irrelevant,

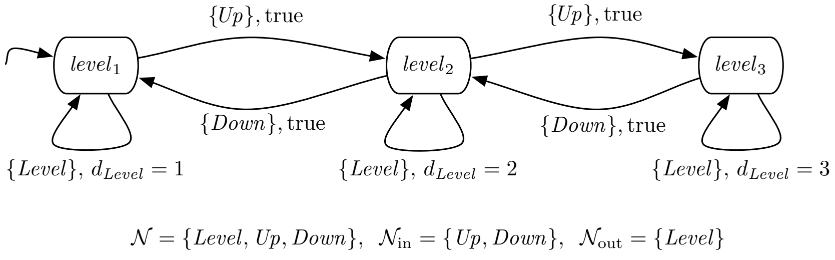
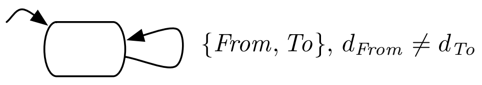

$\{Up\}$, true
$\{Up\}$, true
$level_1$
$level_2$
$level_3$
$\{Down\}$, true
$\{Down\}$, true
$\{Level\}$, $d_{Level} = 1$
$\{Level\}$, $d_{Level} = 2$
$\{Level\}$, $d_{Level} = 3$
$\mathcal{N} = \{Level, Up, Down\}$, $\mathcal{N}_{\text{in}} = \{Up, Down\}$, $\mathcal{N}_{\text{out}} = \{Level\}$

Fig. 3. Constraint automaton for the *Elevator* component (3 levels)



$\mathcal{N} = \{From, To\}, \ \mathcal{N}_{\text{in}} = \varnothing, \ \mathcal{N}_{\text{out}} = \{From, To\}$

Fig. 4. Constraint automaton for a simple variant of request generation

the data constraints at these transitions are always satisfied. For the transitions where the data-flow location *Level* is active, the data constraint ensures that the correct current level of the elevator is transferred.

The users are modelled in a highly abstract way. Fig. 4 shows a constraint automaton where requests are generated non-deterministically and are simultaneously transferred with the current location and the desired destination via the two interface ports. The data constraint ensures that no non-sensical request is generated, where the current location and the destination are the same. □

3 The Coordination Language Reo

Reo [2] is a channel-based, exogenous coordination language. It allows the specification of the coordination glue between components by a network of channels, component connectors and Reo nodes. Reo channels serve as the primitive building blocks for the network. Each channel has two distinct *channel ends*. A channel end can be either a *source end*, through which data enters a channel or a *sink end*, through which data leaves a channel. The operational semantics of Reo networks can be provided in a compositional way using constraint automata for the channels and an appropriate composition operator on constraint automata for the Reo join operation, which joins channel ends together to form Reo nodes in the network.

Reo provides a library of basic channels, which can be extended by user-defined channels. Fig. 5 shows some of the most common channels and their constraint automata representation. The synchronous channel – Fig. 5a – synchronizes its source end and its sink end, transferring the data item from the source end A to the sink end B. The synchronous drain channel – Fig. 5b – has two source ends and synchronizes both of them, consuming both data values. Note that the data constraint in the corresponding constraint automaton does not require both data values to have the same value. The filter channel – Fig. 5c – is an example for a channel with data-dependent behavior. The filter condition is here formalized by a subset $D \subseteq \mathsf{Data}$. If the data value $d \in \mathsf{Data}$ at the source end of the filter channel is acceptable, i.e., $d \in D$, the filter channel behaves like the synchronous channel and passes the data value to the sink end B. Otherwise, the channel blocks. Fig. 5d shows a FIFO1 channel, which can store a single value $d \in \mathsf{Data}$ in its buffer received at the source end, which is then once available for reading at the sink end. The constraint automaton for the FIFO1 channel is depicted here for the data domain $\mathsf{Data} = \{0, 1\}$. The state called $\varnothing$

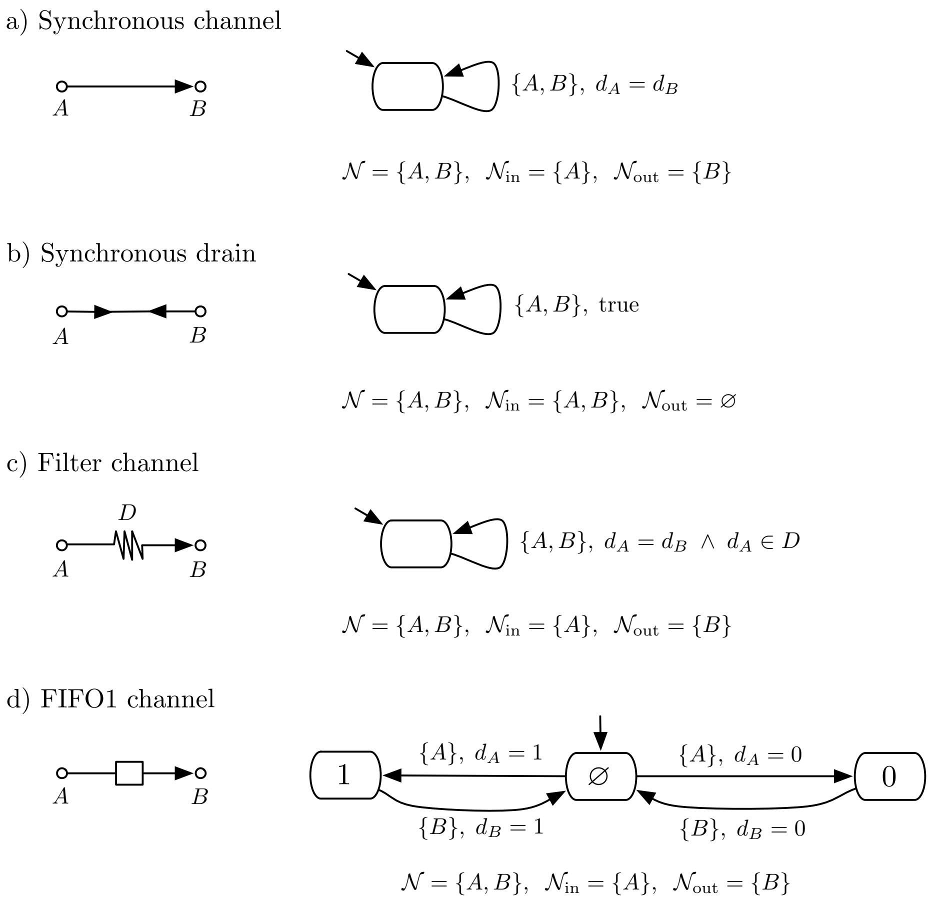

a) Synchronous channel
A
B
$\{A, B\},\ d_A = d_B$
$\mathcal{N} = \{A, B\},\ \mathcal{N}_{\text{in}} = \{A\},\ \mathcal{N}_{\text{out}} = \{B\}$
b) Synchronous drain
A
B
$\{A, B\},\ \text{true}$
$\mathcal{N} = \{A, B\},\ \mathcal{N}_{\text{in}} = \{A, B\},\ \mathcal{N}_{\text{out}} = \varnothing$
c) Filter channel
D
A
B
$\{A, B\},\ d_A = d_B\ \wedge\ d_A \in D$
$\mathcal{N} = \{A, B\},\ \mathcal{N}_{\text{in}} = \{A\},\ \mathcal{N}_{\text{out}} = \{B\}$
d) FIFO1 channel
A
B
1
$\varnothing$
0
$\{A\},\ d_A = 1$
$\{A\},\ d_A = 0$
$\{B\},\ d_B = 1$
$\{B\},\ d_B = 0$
$\mathcal{N} = \{A, B\},\ \mathcal{N}_{\text{in}} = \{A\},\ \mathcal{N}_{\text{out}} = \{B\}$

Fig. 5. Basic Reo channels and the corresponding constraint automaton. For the FIFO1 channel, the constraint automaton is shown for $\mathsf{Data} = \{0, 1\}$.

represents the configuration where the buffer is empty, while the states 0 and 1 represent the configurations where the buffer contains the corresponding data value. The library of channels available as building blocks for the Reo network can be extended by specifying the type of channel ends and the corresponding constraint automaton.

A Reo network arises by joining channel ends at *Reo nodes* which mediate the data flow of all the channel ends coinciding at a node. In this article, we use two variants of nodes: the standard Reo node (depicted as ●) and the *route node* (depicted as ⊗). As an example, consider the Reo network depicted in Fig. 6. The left hand side shows the channels with their channel ends before being connected to the nodes, while the right hand side shows the network after the channel ends have been joined at the nodes.

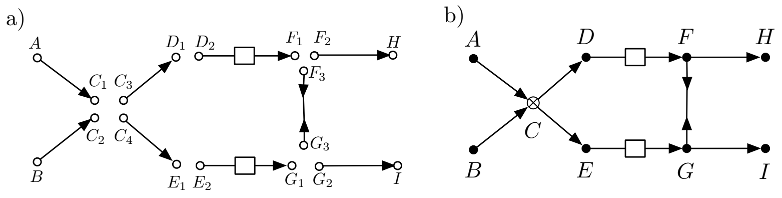


Fig. 6. A Reo network, before (a) and after (b) the channel ends are joined in Reo nodes

Nodes can be classified according to the type of the channel ends connected to them. A node where all the channel ends are source ends is called a *source node.* If all the channel ends are sink ends it is called a *sink node.* A node where both types of channel ends coincide is called a *mixed node.* A mixed Reo node with standard semantics – such as nodes D, E, F, G in Fig. 6b – is active (with data value $d \in \mathsf{Data}$) if both of the following conditions hold:

(S1) Exactly one of the sink ends coinciding at the node is active, with data value d.
(S2) All of the source ends coinciding at the node are active with data value d.

Condition (S1) serves to ensure that a node acts as a non-deterministic merger, choosing exactly one of the channel ends capable of providing data at the moment. Condition (S2) then ensures the replication of the data as the received data value is copied to all the connected source ends simultaneously. Thus, a data item is suitable for selection only if it can be passed on to all the connected source ends.

A mixed node with route semantics – such as node C in Fig. 6b – is active (with data value $d \in \mathsf{Data}$) if both of the following conditions hold:

(R1) Exactly one of the sink ends coinciding at the node is active with data value d.
(R2) Exactly one of the source ends coinciding at the node are active, with data value d.

Condition (R1) is the same as (S1) for the standard Reo nodes, while condition (R2) replaces the replicator semantics of the standard Reo node with a routing semantic, where the received data value is routed to exactly one of the connected source ends.

Source nodes – such as nodes A and B in Fig. 6b – and sink nodes – such as nodes H and I in Fig. 6b – can be regarded as *open* for reading and writing respectively, and serve as the exported interface ports when regarding the Reo network as a component connector. A source node with the standard semantics is active if condition (S2) is satisfied, while a sink node with the standard semantics is active if condition (S1) is satisfied. For source and sink nodes with routing semantics, the same applies with regard to conditions (R2) and (R1) respectively.

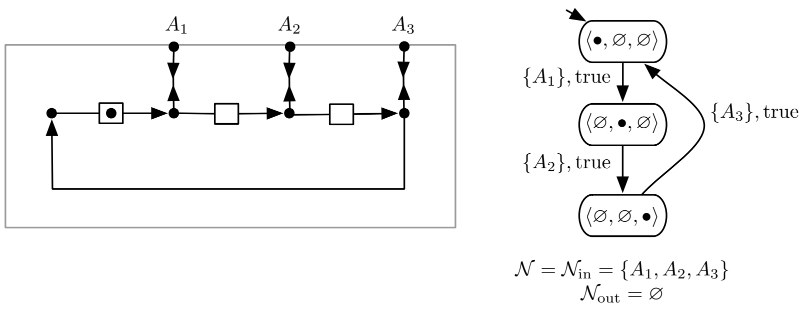

A_1
A_2
A_3
$\langle \bullet, \varnothing, \varnothing \rangle$
$\{A_1\}$, true
$\langle \varnothing, \bullet, \varnothing \rangle$
$\{A_3\}$, true
$\{A_2\}$, true
$\langle \varnothing, \varnothing, \bullet \rangle$
$\mathcal{N} = \mathcal{N}_{\text{in}} = \{A_1, A_2, A_3\}$
$\mathcal{N}_{\text{out}} = \varnothing$

Fig. 7. Component connector realizing a *Sequencer* with three ports and the corresponding constraint automaton where the internal data-flow locations are "hidden" (see Sec. 3.1)

Example 2. We now briefly describe the behavior of the Reo network depicted in Fig. 6b. Data may enter the network at either node A or node B. The route node C ensures that only one of these ports is simultaneously active, as it chooses one of the channel ends C_1 and C_2. The data is routed to one of the channel ends C_3 and C_4 and thus to either node D or E, i.e., one of the two FIFO1 channels if it is able to accept a new data value. Node F can only be active if the upper FIFO1 channel is full, i.e., a data item may be read via channel end F_1, and if the data value can be copied to both the F_2 and F_3 channel ends, i.e., the source end of the synchronous channel to node H and the upper source end of the synchronous drain channel. As a consequence, the synchronous drain channel between nodes F and G ensures that data may only be read from both FIFOs simultaneously. $\square$

Reo networks can be used to coordinate the communication between components connected to it via the interface ports of the components. The interface ports are for this purpose treated just like sink or source channel ends. To facilitate hierarchical modeling, a given Reo network can also be regarded as a component connector with interface ports, which may then be used as a building block in higher-level Reo networks providing a specific coordination pattern. The sink and source nodes of the Reo network of a component connector become the interface ports of the connector. Consider as an example the Reo network realizing a *Sequencer* component connector with three input ports depicted in Fig. 7. The FIFO1 channel on the left is initially filled with a token, which is sequentially passed to the next FIFO1 channel in each step if a data item at the corresponding source node can be consumed. When a component connector such as the *Sequencer* is used at a higher level, the specifics of the internal realization are abstracted ("hidden") and it becomes a basic building block providing a specific coordination behavior at its interface ports. Component connectors may also be specified directly by providing their behavior as a constraint automaton, just like user defined channels. Channels can be regarded as simple component connectors with exactly two interface ports.

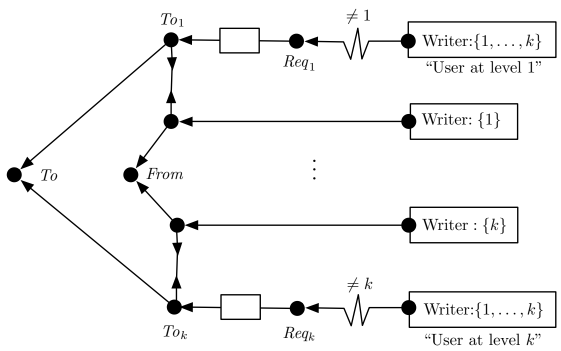


Fig. 8. Reo network for buffered request generation for k levels

Example 3. As a further example, consider the Reo network in Fig. 8, which depicts a refinement of the *Requests* component in the elevator system. In this network, requests are initiated by one of the components representing the user located at a certain level ("User at level i"), modeled by a simple *Writer* component. These components produce data items non-deterministically chosen from a set of data items, in this case from the potential destination levels. A filter channel then blocks those destination choices that would be non-sensical, i.e., where the user is at level i and requests to go to the same destination i. The request is transferred via node Req_i into a FIFO1 channel. Whenever a new request is needed by the *Controller* via the *To* interface port, one of the full FIFO1 channels is non-deterministically chosen and the stored data item is output via *To*. The synchronous drain channels then ensure that simultaneously the corresponding current level of the request is provided by the *Writer* at the corresponding level and transferred via the interface port *From*. □

3.1 Constraint Automata as the Operational Semantics for Reo

Given constraint automata for all parts of a Reo network, i.e., for the channels and component connectors, for the nodes and the components, it is possible to compositionally build a constraint automaton representing the composite system [10]. The construction relies on the parallel composition of the constraint automata of all parts of the system using an appropriate product automata construction for constraint automata with synchronous data-flow at shared data-flow locations.

Definition 5 (Product automaton). The product automaton of two constraint automata

$$\mathcal{A}_1 = (Q_1, \mathcal{N}_1, \mathcal{N}_{\text{in}}^1, \mathcal{N}_{\text{out}}^1, \longrightarrow_1, Q_{0,1}) \quad \text{and} \quad \mathcal{A}_2 = (Q_2, \mathcal{N}_2, \mathcal{N}_{\text{in}}^2, \mathcal{N}_{\text{out}}^2, \longrightarrow_2, Q_{0,2})$$

is the constraint automaton

$$\mathcal{A}_1 \bowtie \mathcal{A}_2 = (Q_1 \times Q_2,\ \mathcal{N}_1 \cup \mathcal{N}_2,\ \mathcal{N}_{\text{in}},\ \mathcal{N}_{\text{out}},\ \longrightarrow,\ Q_{0,1} \times Q_{0,2})$$

where the transition relation $\longrightarrow$ of the product constraint automaton is defined by the following rules:

$$\frac{q \xrightarrow{N_1,g_1}_1 q' \;\wedge\; p \xrightarrow{N_2,g_2}_2 p' \;\wedge\; N_1 \cap \mathcal{N}_2 = N_2 \cap \mathcal{N}_1}{\langle q,p\rangle \xrightarrow{N_1 \cup N_2, g_1 \wedge g_2} \langle q',p'\rangle}$$

$$\frac{q \xrightarrow{N_1,g_1}_1 q' \;\wedge\; N_1 \cap \mathcal{N}_2 = \varnothing}{\langle q,p\rangle \xrightarrow{N_1,g_1} \langle q',p\rangle} \qquad \frac{p \xrightarrow{N_2,g_2}_2 p' \;\wedge\; N_2 \cap \mathcal{N}_1 = \varnothing}{\langle q,p\rangle \xrightarrow{N_2,g_2} \langle q,p'\rangle}$$

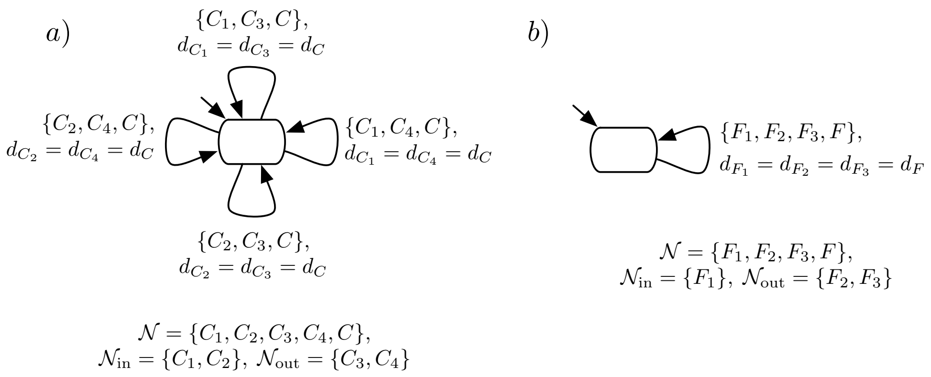
The first rule for the transition relation for the product automaton handles the case that both automata perform a step at the same time, requiring that the active data-flow locations shared by both automata agree. The second and the third rule handle the case that only one of the automata performs a step, which may only happen if none of the active data-flow locations is shared with the other automaton.

We require that for all data-flow locations $A \in \mathcal{N}_1 \cap \mathcal{N}_2$ shared between $\mathcal{A}_1$ and $\mathcal{A}_2$, either $A \in \mathcal{N}^1_{\text{in}}$ and $A \in \mathcal{N}^2_{\text{out}}$ or $A \in \mathcal{N}^1_{\text{out}}$ and $A \in \mathcal{N}^2_{\text{in}}$, i.e., that only appropriate data-flow locations are "plugged together". The sets $\mathcal{N}_{\text{in}}$ and $\mathcal{N}_{\text{out}}$ in the product automaton then consist of the remaining data-flow locations that are not shared, i.e.,

$$\mathcal{N}_{\text{in}} = (\mathcal{N}^1_{\text{in}} \cup \mathcal{N}^2_{\text{in}}) \setminus (\mathcal{N}_1 \cap \mathcal{N}_2) \quad \text{and} \quad \mathcal{N}_{\text{out}} = (\mathcal{N}^1_{\text{out}} \cup \mathcal{N}^2_{\text{out}}) \setminus (\mathcal{N}_1 \cap \mathcal{N}_2).$$

□

The compositional approach for generating the constraint automaton for a Reo network relies on the assumption that we are given separate constraint automata for all channels in the Reo network and for the component connectors and components. The application of the product construction given in Def. 5 requires some renaming of the data-flow locations. In the constraint automata for the channels, the data-flow locations are renamed to the corresponding names used in the Reo network. E.g., the constraint automaton for the synchronous channel with channel ends C_3 and D_1 in Fig. 6a is obtained from the generic constraint automaton for the synchronous channel depicted in Fig. 5a by renaming the data-flow location A to C_3 and renaming B to D_1. Similarly, such a renaming is applied to the constraint automata for component connectors and components to map their interface ports to the names used in the Reo network. We assume here that the names used in the Reo network for the channel ends and imported interface ports are distinct. For all Reo nodes in the network we construct appropriate constraint automata that ensure the correct node semantics for the coinciding channel ends. As an example, Fig. 9 shows the constraint automata for the nodes C and F in Fig. 6b. The constraint automata for the nodes share data-flow locations with the coincident channel ends and can thus coordinate the data flow for the connected channels. The constraint automaton for the composite system is then the product automaton of all the constraint automata for the channels, component connectors, nodes and components.



$a)$
$\{C_1, C_3, C\}$, $d_{C_1} = d_{C_3} = d_C$
$\{C_2, C_4, C\}$, $d_{C_2} = d_{C_4} = d_C$
$\{C_1, C_4, C\}$, $d_{C_1} = d_{C_4} = d_C$
$\{C_2, C_3, C\}$, $d_{C_2} = d_{C_3} = d_C$
$\mathcal{N} = \{C_1, C_2, C_3, C_4, C\}$, $\mathcal{N}_{\text{in}} = \{C_1, C_2\}$, $\mathcal{N}_{\text{out}} = \{C_3, C_4\}$
$b)$
$\{F_1, F_2, F_3, F\}$, $d_{F_1} = d_{F_2} = d_{F_3} = d_F$
$\mathcal{N} = \{F_1, F_2, F_3, F\}$, $\mathcal{N}_{\text{in}} = \{F_1\}$, $\mathcal{N}_{\text{out}} = \{F_2, F_3\}$

Fig. 9. Constraint automata for a) the route node C and b) the standard Reo node F in Fig. 6

Hiding. The hide operator for constraint automata provides a means to declare certain data-flow locations as local and non-observable from outside. This can, e.g., be used to abstract from internal behavior of a component connector or Reo network or remove information about the channel ends.

Definition 6 (Hiding). Let $\mathcal{A} = (Q, \mathcal{N}, \mathcal{N}_{\text{in}}, \mathcal{N}_{\text{out}}, \longrightarrow_{\mathcal{A}}, Q_0)$ be a constraint automaton. The result of hiding a data-flow location $A \in \mathcal{N}$ from $\mathcal{A}$ is the constraint automaton

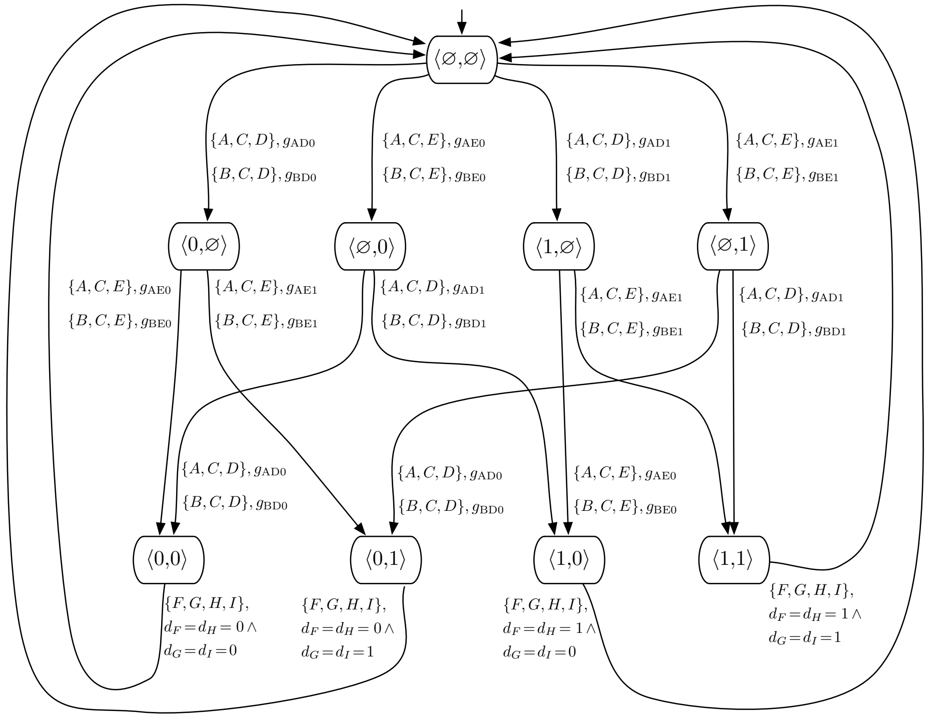
$$\mathcal{A}' = (Q,\ \mathcal{N} \setminus \{A\},\ \mathcal{N}_{\text{in}} \setminus \{A\},\ \mathcal{N}_{\text{out}} \setminus \{A\},\ \longrightarrow_{\mathcal{A}'}, Q_0).$$

The transition relation $\longrightarrow_{\mathcal{A}'}$ is given by:

$$\frac{q \xrightarrow{N,g}_{\mathcal{A}} p}{q \xrightarrow{N \setminus \{A\}, \exists[A]g}_{\mathcal{A}'} p} \qquad \text{where } \exists[A]g = \bigvee_{d \in \mathsf{Data}} g[d_A/d].$$

Here, we write $g[d_A/d]$ to denote the data constraint obtained from g by syntactically replacing all occurrences of atoms of the form "$d_A = d'$" by "true" if $d = d'$ and by "$\neg$true" if $d \neq d'$, as well as replacing the atoms of the form "$d_A = d_B$" for some $B \in \mathcal{N}$ with "$d_B = d$". $\square$

As an example, reconsider the Reo network in Fig. 6b. The corresponding product automaton for this Reo network is depicted in Fig. 10, after hiding the channel ends and for the data domain $\mathsf{Data} = \{0, 1\}$. The state space of the constraint automaton consists of the Cartesian product of the local state spaces of the two FIFO1 channels in the network. Recall that the states of a FIFO1 channel for this data domain are $\varnothing$ if the buffer is empty or 0 or 1 if the buffer is full and contains that value.



$g_{\mathrm{AD0}}: \ d_A = d_C = d_D = 0$	$g_{\mathrm{AD1}}: \ d_A = d_C = d_D = 1$
$g_{\mathrm{BD0}}: \ d_B = d_C = d_D = 0$	$g_{\mathrm{BD1}}: \ d_B = d_C = d_D = 1$
$g_{\mathrm{AE0}}: \ d_A = d_C = d_E = 0$	$g_{\mathrm{AE1}}: \ d_A = d_C = d_E = 1$
$g_{\mathrm{BE0}}: \ d_B = d_C = d_E = 0$	$g_{\mathrm{BE1}}: \ d_B = d_C = d_E = 1$

Fig. 10. Constraint automaton for the Reo network in Fig. 6, with $\mathsf{Data} = \{0, 1\}$, $\mathcal{N} = \{A, B, C, D, E, F, G, H, I\}$, $\mathcal{N}_{\mathrm{in}} = \{A, B\}$ and $\mathcal{N}_{\mathrm{out}} = \{H, I\}$

4 Verification of Components and Connectors

Constraint automata yield a general framework for the behavior of a component, a connector or a composite system and serve as starting points for model checking. The model checking problem asks whether a given property holds for a given automaton. In this framework and the tool Vereofy (see Sec. 4.5), the properties can be specified by temporal formulas with classical modalities to formalize safety or liveness conditions, but also constraints on the observable data flow (I/O-streams). Vereofy supports model checking against linear-time, branching-time properties formalized in the logics $\mathtt{LTL}_{\mathrm{IO}}$ or $\mathtt{BTSL}$. The logic $\mathtt{LTL}_{\mathrm{IO}}$ (see Sec. 4.2) is a variant of linear temporal logic LTL which is closely related to dynamic LTL [22] and combines the standard temporal modalities of LTL with stream expressions. It is appropriate to specify complex temporal conditions on paths (such as Boolean combinations of reachability, repeated reachability or persistence conditions), possibly in combination with regular conditions on I/O-streams of their prefixes. The logic BTSL (see Sec. 4.3) combines features of CTL [16,17], PDL [19] and timed data stream logic (TDSL) [3,15]. The standard CTL-operators are combined with special path modalities that allow reasoning about the data streams observable at data-flow locations by means of stream expressions. Section 4.4 outlines the main concepts for checking equivalence based on bisimulations.

4.1 Atomic Propositions, I/O Constraints and Stream Expressions

Our logical framework to specify properties of systems where the component interfaces are modeled by constraint automata and their glue code by a Reo network relies on features that allow to express conditions on 1) the local states of components or component connectors and 2) the I/O-activity at the data-flow locations. For the states, we employ the standard concept of atomic propositions, while for the I/O-activity we use I/O-constraints and stream expressions, i.e., regular expressions over the I/O-activity.

Atomic Propositions are used to express primitive statements on the states of the system under consideration. Given a finite set AP of atomic propositions, a labeling function $L : Q \to 2^{\mathsf{AP}}$ assigns to each state $q \in Q$ in the constraint automaton a set of atomic propositions $L(q) \subseteq \mathsf{AP}$ that are satisfied in q.

Definition 7 (Labeled Constraint Automaton)
A labeled constraint automaton is a tuple $\mathcal{A} = (Q, \mathcal{N}, \mathcal{N}_{\text{in}}, \mathcal{N}_{\text{out}}, \longrightarrow, Q_0, \mathsf{AP}, L)$ where Q, $\mathcal{N}$, $\mathcal{N}_{\text{in}}$, $\mathcal{N}_{\text{out}}$, $\longrightarrow$ and Q_0 are as in Def. 2 and where

– AP is a finite set of atomic propositions and
– $L : Q \to 2^{\mathsf{AP}}$ is a labeling function. □

Example 4. As an example, consider the constraint automaton for the FIFO1 channel in Fig. 5d and the set of atomic propositions

$$\mathsf{AP} = \{\texttt{"buffer is empty"}, \texttt{"buffer is full"}, \texttt{"buffer contains 0"}\}.$$

The labeling function then maps the initial state $\varnothing$ representing the empty FIFO buffer to the set {`"buffer is empty"`}. Furthermore, it maps the state 0 to the set {`"buffer is full"`, `"buffer contains 0"`} and maps state 1 to {`"buffer is full"`}. □

I/O-constraints and Stream Expressions. To be able to concisely formalize sets of I/O-activity, we extend the concept of data constraints to I/O-constraints, propositional formulas over the activity and the data items at data-flow locations.

Definition 8 (I/O-constraints). The abstract syntax of *I/O-constraints* over the set $\mathcal{N}$ of data-flow locations is given by the grammar:

$$\mathsf{ioc} ::= \ tt \ \big| \ A \ \big| \ d_A = d \ \big| \ d_A = d_B \ \big| \ \neg\mathsf{ioc} \ \big| \ \mathsf{ioc}_1 \vee \mathsf{ioc}_2$$

where $A, B \in \mathcal{N}$ and $d \in \mathsf{Data}$. Each I/O-constraint ioc stands for a set of concurrent I/O-operations $\|\mathsf{ioc}\| \subseteq \mathsf{CIO}_{\mathcal{N}}$, defined as follows:

$$\begin{array}{rcl} \|tt\| & \overset{\text{def}}{=} & \mathsf{CIO}_{\mathcal{N}} \\ \|A\| & \overset{\text{def}}{=} & \{ c \in \mathsf{CIO}_{\mathcal{N}} \,:\, A \in \mathit{active}(c) \} \\ \|d_A = d\| & \overset{\text{def}}{=} & \{ c \in \mathsf{CIO}_{\mathcal{N}} \,:\, A \in \mathit{active}(c) \wedge c(A) = d \} \\ \|d_A = d_B\| & \overset{\text{def}}{=} & \{ c \in \mathsf{CIO}_{\mathcal{N}} \,:\, A, B \in \mathit{active}(c) \wedge c(A) = c(B) \} \\ \|\neg \mathsf{ioc}\| & \overset{\text{def}}{=} & \mathsf{CIO}_{\mathcal{N}} \setminus \|ioc\| \\ \|\mathsf{ioc}_1 \vee \mathsf{ioc}_2\| & \overset{\text{def}}{=} & \|ioc_1\| \cup \|ioc_2\| \end{array}$$

□

As for the data constraints, we derive the standard propositional operators and syntactic shorthand notations for data constraints. The notation $\{A_1, \ldots, A_n\}$ with $A_1, \ldots, A_n \in \mathcal{N}$, signifying that exactly the data-flow locations $A_1, \ldots, A_n$ are active, is used as a shorthand for

$$\bigwedge_{A \in N} A \quad \wedge \quad \bigwedge_{B \in \mathcal{N} \setminus N} \neg B,$$

where $N = \{A_1, \ldots, A_n\}$, i.e.,

$$\|\{A_1, \ldots, A_n\}\| = \{ c \in \mathsf{CIO}_{\mathcal{N}} : \mathit{active}(c) = \{A_1, \ldots, A_n\} \}.$$

To impose conditions on the data flow at the I/O-ports of components or nodes in the network, our logics use a symbolic representation for sets of I/O-streams by means of regular I/O-stream expressions, briefly called stream expressions.

Definition 9 (Stream expression). The abstract syntax of stream expressions over $\mathcal{N}$ is given by the following grammar:

$$\alpha \quad ::= \quad \mathsf{ioc} \;\Big|\; \alpha^* \;\Big|\; \alpha_1 ; \alpha_2 \;\Big|\; \alpha_1 \cup \alpha_2$$

where ioc ranges over all I/O-constraints over $\mathcal{N}$. The formal definition of the regular languages $\mathsf{IOS}(\alpha) \subseteq \mathsf{CIO}^*$ is defined by structural induction. $\mathsf{IOS}(\mathsf{ioc})$ is the set consisting of the I/O-streams of length 1 given by ioc, i.e., $\mathsf{IOS}(\mathsf{ioc}) = \|\mathsf{ioc}\|$. Union ($\cup$), Kleene star (*) and concatenation (;) have their standard meaning as in ordinary regular expressions. □

4.2 Linear-Time Properties: $\mathsf{LTL_{IO}}$

In this section we describe the logic $\mathsf{LTL_{IO}}$ [8], which is adapted from Dynamic Linear Time Temporal Logic (DLTL) [22] for the context of constraint automata and I/O-stream expressions. DLTL itself extends LTL with regular expressions to achieve the full expressiveness of ω-regular languages. For $\mathsf{LTL_{IO}}$ the concurrent I/O-operations over the set $\mathcal{N}$ (i.e., the elements in the set $\mathsf{CIO}_{\mathcal{N}}$) serve as names for actions and the I/O-stream expressions take the role of the propositional dynamic logic programs (regular expressions) of DLTL.

Definition 10 (Syntax of $\mathtt{LTL_{IO}}$). The abstract syntax of $\mathtt{LTL_{IO}}$ formulas over AP and $\mathcal{N}$ is defined by the following grammar.

$$\varphi \ ::= \ \text{true} \ \Big| \ a \ \Big| \ \neg\varphi \ \Big| \ \varphi_1 \wedge \varphi_2 \ \Big| \ \varphi_1 \, \mathtt{U}^{\alpha} \, \varphi_2$$

where $a \in \mathsf{AP}$ and α is a stream expression over $\mathcal{N}$ as in Def. 9. □

Definition 11 (Semantics of $\mathtt{LTL_{IO}}$)
Let $\mathcal{A} = (Q, \mathcal{N}, \mathcal{N}_{\text{in}}, \mathcal{N}_{\text{out}}, \longrightarrow, Q_0, \mathsf{AP}, L)$ be a labeled constraint automaton and let φ be an $\mathtt{LTL_{IO}}$ formula over AP and $\mathcal{N}$. Given a path $\pi = q_0 \xrightarrow{c_1} q_1 \xrightarrow{c_2} \ldots$ in $\mathcal{A}$, the satisfaction relation $\pi \models \varphi$ is defined as follows:

$$\begin{array}{lll}
\pi \models \text{true} & & \\
\pi \models a & \text{iff} & a \in L(q_0) \\
\pi \models \neg\varphi & \text{iff} & \pi \not\models \varphi \\
\pi \models \varphi_1 \wedge \varphi_2 & \text{iff} & \pi \models \varphi_1 \text{ and } \pi \models \varphi_2 \\
\pi \models \varphi_1 \, \mathtt{U}^{\alpha} \, \varphi_2 & \text{iff} & \text{there exists } n \geq 0 \text{ such that } \pi{\uparrow}n \models \varphi_2 \text{ and} \\
 & & \mathsf{ios}(\pi{\downarrow}n) \in \mathsf{IOS}(\alpha) \text{ and } \pi{\uparrow}i \models \varphi_1 \text{ for all } 0 \leq i < n
\end{array}$$

□

Recall that $\pi{\downarrow}n$ is the prefix of π of length n, $\pi{\uparrow}n$ is the suffix of π starting at the n-th state, $\mathsf{ios}(\eta)$ is the projection on the corresponding I/O-stream and $\mathsf{IOS}(\alpha)$ is the set of I/O-streams satisfying the stream expression α.

The until operator is indexed by a stream expression α over I/O-constraints. Intuitively, it is satisfied on a given path if there exists a finite prefix such that its I/O-stream satisfies α and φ_1 holds for all the suffixes starting at a state in this prefix and φ_2 holds for the suffix starting in the state after the prefix matching α. In addition to the usual propositional operators ($\vee$, $\rightarrow$, $\leftrightarrow$, etc.) we can derive the path modalities $\langle\!\langle\alpha\rangle\!\rangle\varphi$ and $[\![\alpha]\!]\varphi$ by

$$\langle\!\langle\alpha\rangle\!\rangle\varphi \stackrel{\text{def}}{=} \text{true}\,\mathtt{U}^{\alpha}\,\varphi \quad \text{and} \quad [\![\alpha]\!]\varphi \stackrel{\text{def}}{=} \neg\langle\!\langle\alpha\rangle\!\rangle\neg\varphi.$$

Intuitively, $\langle\!\langle\alpha\rangle\!\rangle\varphi$ holds if there *exists a prefix* whose I/O-stream matches α and afterwards φ holds for the suffix. The dual operator, $[\![\alpha]\!]\varphi$, holds if *for all prefixes* with I/O-streams matching α afterwards φ holds for the suffix.

The standard LTL until operator without stream expressions can be derived by $\varphi_1 \, \mathtt{U} \, \varphi_2 \stackrel{\text{def}}{=} \varphi_1 \, \mathtt{U}^{tt^*} \, \varphi_2$, where tt^* is the stream expression signifying an I/O-stream of any finite length. We can derive as well the standard LTL operators "eventually $\Diamond$", "always $\Box$" and "neXt $\mathtt{X}$":

$$\Diamond\,\varphi \stackrel{\text{def}}{=} \text{true}\,\mathtt{U}\,\varphi, \quad \Box\,\varphi \stackrel{\text{def}}{=} \neg\Diamond\neg\varphi, \quad \mathtt{X}\,\varphi \stackrel{\text{def}}{=} \langle\!\langle tt\rangle\!\rangle\varphi.$$

Given a constraint automaton $\mathcal{A}$, the model checking problem asks whether all paths in $\mathcal{A}$ starting in an initial state satisfy the formula φ:

$$\mathcal{A} \models \varphi \quad \stackrel{\text{def}}{\Longleftrightarrow} \quad \pi \models \varphi \text{ for all } \pi \in \mathsf{Paths}(q_0) \text{ and all } q_0 \in Q_0$$

Example 5. We will now provide some example formulas for the elevator system with k levels as described in the previous sections. We start with a formula specifying for the *Elevator* component (cf. Fig. 3) that the elevator will not move if there is no *Up* or *Down* command:

$$\varphi_1 = \bigwedge_{1 \le i \le k} \Big(\Box \left(\texttt{"elevator at } i\texttt{"} \rightarrow [\![(\neg \mathit{Up} \wedge \neg \mathit{Down})^*]\!] \texttt{"elevator at } i\texttt{"} \right) \Big)$$

Here, the atomic propositions `"elevator at` i`"` characterize all states where the elevator is at level i. For the example automaton in Fig. 3, the labeling function L maps state level_i to $L(\mathit{level}_i) = \{\texttt{"elevator at } i\texttt{"}\}$. The subformula

$$[\![(\neg \mathit{Up} \wedge \neg \mathit{Down})^*]\!] \texttt{"elevator at } i\texttt{"}$$

holds if, for all paths with an I/O-stream prefix that does not contain an active *Up* or *Down*, the elevator is still at level i. This includes the empty I/O-stream prefix, where no I/O occurred. The whole formula φ_1 thus holds if for all levels i it is always the case that whenever the elevator is at level i and some arbitrary I/O occurs without *Up* or *Down* commands the elevator will still be at the same level i.

The next formula specifies for the composite system that whenever the *Controller* receives a user request it will be the case that the *Elevator* services that request:

$$\varphi_2 = \bigwedge_{\substack{1 \le i \le k \\ 1 \le j \le k}} \Big(\Box \big(\langle\!\langle \mathit{From} \wedge \mathit{To} \wedge d_{\mathit{From}} = i \wedge d_{\mathit{To}} = j \rangle\!\rangle \mathrm{true} \rightarrow \Diamond (\texttt{"elevator at } i\texttt{"} \wedge \Diamond \texttt{"elevator at } j\texttt{"}) \big) \Big)$$

The left hand side of the implication holds whenever a user request is received by the *Controller* via the *From* and *To* ports. The right hand side of the implication holds if eventually the elevator visits the level i where the request originated and then later on eventually visits the requested destination level j.

As a further example, consider the elevator system depicted in Fig. 11 with two elevators. We assume here that both elevators share the same elevator shaft. It thus becomes imperative that both elevators do not crash into each other, i.e., are at the same level. This property can be specified by the following formula:

$$\varphi_3 = \bigwedge_{1 \le i \le k} \Big(\Box \left(\texttt{"elevator A at level } i\texttt{"} \rightarrow \neg \texttt{"elevator B at level } i\texttt{"} \right) \Big)$$

The *Controller* has to insure that φ_3 always holds. Alternatively, it would also be possible to ensure this property by inserting a Reo network between the *Controller* and the elevators that provides appropriate coordination for the elevator commands and blocks those that would lead to an elevator crash. □

Model Checking $\mathtt{LTL}_{\mathrm{IO}}$ **Formulas.** To determine whether $\mathcal{A} \models \varphi$, the standard automata theoretic approach to LTL model checking [32,31] can be used, as illustrated in Fig. 12. For an $\mathtt{LTL}_{\mathrm{IO}}$ formula ϕ, first construct a non-deterministic

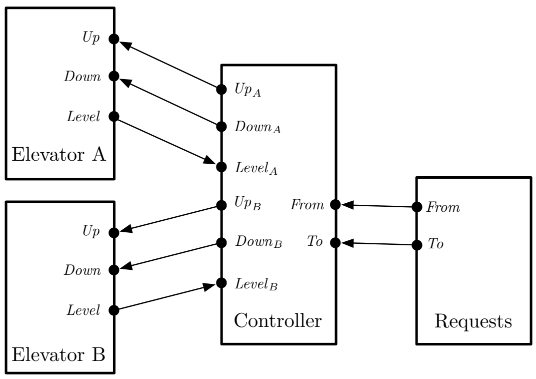


Fig. 11. Elevator system with 2 elevators, sharing the same elevator shaft

Büchi automaton recognizing exactly the paths $\pi \models \phi$ (e.g., using the construction in [21]). Non-deterministic Büchi automata (NBA) are similar to non-deterministic finite automata over finite words, but range over infinite words. The Büchi acceptance condition specifies a subset of automata states that has to be visited infinitely often for a path to be accepted. To check whether $\mathcal{A} \models \varphi$, we construct a non-deterministic Büchi automaton $\mathcal{Z}_{\neg\varphi}$ for the negation of φ. $\mathcal{Z}_{\neg\varphi}$ recognizes all the paths that violate φ. Then the product automaton $\mathcal{A} \bowtie \mathcal{Z}_{\neg\varphi}$ is built, resulting in a constraint automaton augmented with a Büchi acceptance condition. The paths of the product can be viewed as pairs $\langle \pi, s \rangle$ of a path π in $\mathcal{A}$ and a run s for π in $\mathcal{Z}_{\neg\varphi}$. The model checking algorithm then searches a path in the product such that s meets the acceptance condition of $\mathcal{Z}_{\neg\varphi}$. If such a path $\langle \pi, s \rangle$ exists then π is a path in $\mathcal{A}$ that violates φ. Otherwise no such path in $\mathcal{A}$ violates φ and consequently $\mathcal{A} \models \varphi$. In the case that $\mathcal{A} \not\models \varphi$, the path in $\mathcal{A}$ violating φ can be output to the user as a counterexample to φ. This allows the user to inspect the model and find the cause of the property violation.

Complexity. The complexity of $\mathtt{LTL}_{\mathrm{IO}}$ model checking depends on two factors, the size of the constraint automaton and the length of the formula. The non-deterministic Büchi automaton generated by the algorithm of [21] for a given formula has in the worst-case an exponentially larger number of states compared to the length of the formula. The search for an accepting path in the product of the constraint automaton and the Büchi automaton can then be carried out in linear time of the size of the product automaton. In practice, the state space explosion problem tends to be the limiting factor for model checking, as most formulas used in practice can be translated to Büchi automata of reasonable size by using optimized algorithms that avoid the potential exponential blowup in many cases.

Fairness. It is often useful to restrict the behavior of an interleaving model to those paths that satisfy some fairness constraints, e.g., to rule out infinite behavior that is considered unrealistic as the activities of some components are

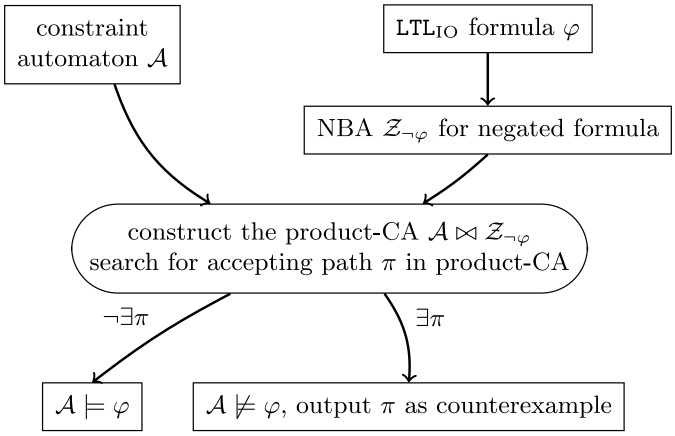

constraint automaton $\mathcal{A}$
$\mathtt{LTL}_{\mathrm{IO}}$ formula φ
NBA $\mathcal{Z}_{\neg\varphi}$ for negated formula
construct the product-CA $\mathcal{A} \bowtie \mathcal{Z}_{\neg\varphi}$
search for accepting path π in product-CA
$\neg\exists\pi$
$\exists\pi$
$\mathcal{A} \models \varphi$
$\mathcal{A} \not\models \varphi$, output π as counterexample

Fig. 12. Schema for model checking an $\mathtt{LTL}_{\mathrm{IO}}$ formula

ignored forever. In the classical approach (e.g., [20]), fairness is used to rule out exceptional or "unrealistic" behaviors. Most prominent is process fairness that serves to discard interleavings that are "unfair" for some processes. Suppose we are given a parallel system with processes $P_1, \ldots, P_n$, then the standard variants of fairness assumptions can be formalized as LTL formulas

– Unconditional fairness: $\Box\Diamond$ "engaged(P_i)"
– Weak fairness: $\Diamond\Box$ "enabled(P_i)" $\rightarrow \Box\Diamond$ "engaged(P_i)"
– Strong fairness: $\Box\Diamond$ "enabled(P_i)" $\rightarrow \Box\Diamond$ "engaged(P_i)"

where "enabled(P_i)" characterizes states where process P_i may by scheduled and "engaged(P_i)" asserts that process P_i has actually been scheduled. Unconditional fairness requires that process P_i is infinitely often scheduled, while weak fairness requires that, if a process is from some point on continually enabled, it will be scheduled infinitely often. Strong fairness requires that if a process can be infinitely often that it will be actually scheduled infinitely often.

In the context of Reo and constraint automata, it is natural to base the fairness assumptions on the I/O-behavior of nodes, e.g., that a Reo node chooses between its possible inputs in a fair way. Atomic propositions enabled(ioc) (where ioc is an I/O-constraint) will be used to assert the enabledness of some concurrent I/O-operation that satisfies ioc. That is, we suppose that enabled(ioc) is an atomic proposition such that the labeling function L enjoys the following property:

$$\mathtt{enabled}(\mathsf{ioc}) \in L(q) \quad \Leftrightarrow \quad \exists\, q \xrightarrow{N,g} q' :\ \mathsf{CIO}(N,g)\ \cap\ \|\mathsf{ioc}\| \neq \varnothing$$

Recall that $\mathsf{CIO}(N,g)$ consists of all concurrent I/O-operations consistent with the transition label N, g and $\|\mathsf{ioc}\|$ consists of all concurrent I/O-operations consistent with the I/O-constraint ioc.

For example, the fairness assumptions regarding the activity at some Reo node A can then be formalized as follows:

- Unconditional fairness: $\Box\Diamond\langle\!\langle A\rangle\!\rangle\mathrm{true}$
- Weak fairness: $\Diamond\Box\,\mathtt{enabled}(A) \rightarrow \Box\Diamond\langle\!\langle A\rangle\!\rangle\mathrm{true}$
- Strong fairness: $\Box\Diamond\,\mathtt{enabled}(A) \rightarrow \Box\Diamond\langle\!\langle A\rangle\!\rangle\mathrm{true}$

The unconditional fairness condition requires that node A will be active infinitely often, the weak fairness condition requires that, if there is from some point on continually the possibility of node A being active, that A will be active infinitely often. The strong fairness condition requires that if A can be active infinitely often it will be active infinitely often. The flexibility of the I/O-constraints allows the formalization of other, more complex fairness assumptions for the I/O-behavior, including data dependent fairness conditions. Fairness assumptions corresponding to process fairness for the Reo framework, e.g., that the activity of a certain component is scheduled in a fair way, can be formalized as well by using I/O-constraints that capture all the activity of the component.

Model checking an $\mathtt{LTL}_{\mathrm{IO}}$ formula φ under fairness constraints $\psi_1, \ldots, \psi_k$, where the ψ_i's are $\mathtt{LTL}_{\mathrm{IO}}$ formulas, can be performed by checking the formula $\varphi' = (\psi_1 \wedge \ldots \wedge \psi_k) \rightarrow \varphi$ or by using algorithms adapted to take the fairness assumptions directly into account.

Example 6. Consider as an example the elevator system where the *Requests* component is realized by the Reo network depicted in Fig. 8. This allows for the distinction between a request being made by a user at level i, i.e., node Req_i is active, and the request being transferred to the *Controller*, i.e., nodes To_i and To are active.

We can then adapt the formula φ_2 from Example 5 to specify that whenever a user request is made (in contrast to the request being received by the *Controller* as in φ_2) that the elevator will eventually service that request:

$$\varphi_4 = \bigwedge_{\substack{1\le i\le k\\ 1\le j\le k}} \Big(\Box\,\big(\langle\!\langle Req_i \wedge d_{Req_i} = j\rangle\!\rangle\mathrm{true} \rightarrow \Diamond(\texttt{"elevator at } i\texttt{"} \wedge \Diamond\,\texttt{"elevator at } j\texttt{"})\big)\Big)$$

Even assuming that the *Controller* correctly services the request once it becomes aware of it, i.e., formula φ_2 holds, formula φ_4 will in general not be satisfied. It can be the case that, e.g., a user at level 3 first requests the elevator, with another user at level 2 subsequently also requesting it. The non-deterministic choice at the Reo node To is then resolved to transfer the request for the user at level 2 to the *Controller*. While the *Controller* and elevator services this request originating at level 2, another user at the same level again requests the elevator, and the non-deterministic choice at node To is again resolved in favor of that user. This may happen infinitely often, and thus, the buffered user request originating at level 3 is continuously ignored.

We can introduce a fairness assumption that disallows such unfair, unrealistic behavior, forcing the Reo node To to resolve the non-determinism in a fair way:

$$\psi_{\mathrm{fair(To)}} = \bigwedge_{1\le i\le k} \Big(\Box\Diamond\,\mathtt{enabled}(To_i) \;\rightarrow\; \Box\Diamond\langle\!\langle To_i\rangle\!\rangle\mathrm{true}\Big)$$

This strong fairness condition requires that whenever node *To*$_i$ is enabled infinitely often, i.e., there is a request buffered in the corresponding FIFO1 channel, that then it is active infinitely often. Under the fairness assumption $\psi_{\mathrm{fair(To)}}$ and assuming that the *Controller* correctly services the requests it receives, it can then be shown that φ_4 holds. □

4.3 Branching Time Stream Logic

In this section we introduce a branching time temporal logic for reasoning about the control and data flow of a constraint automaton. The logic, called Branching Time Stream Logic (BTSL) [26], combines features of CTL [16,17], PDL [19] and timed data stream logic (TDSL) [3,15]. As in CTL, formulas may refer to the configurations of a component connector (states of a constraint automaton) by means of atomic propositions $a \in \mathsf{AP}$ and may use the path quantifiers $\exists$ and $\forall$.

Definition 12 (Syntax of BTSL). The abstract syntax of BTSL over AP and $\mathcal{N}$ is given by the following abstract grammar for state formulas Φ and path formulas φ:

$$\begin{aligned}\Phi &:= \mathrm{true} \;\Big|\; a \;\Big|\; \Phi_1 \wedge \Phi_2 \;\Big|\; \neg\Phi \;\Big|\; \exists\varphi \;\Big|\; \forall\varphi \\ \varphi &:= \Phi_1 \,\mathtt{U}^{\alpha}\, \Phi_2\end{aligned}$$

where $a \in \mathsf{AP}$ and α is a stream expression over $\mathcal{N}$ as in Def. 9. □

Derived path modalities. The path modalities $\langle\!\langle\alpha\rangle\!\rangle\Phi$ and $[\![\alpha]\!]\Phi$ can be derived by $\langle\!\langle\alpha\rangle\!\rangle\Phi \overset{\mathrm{def}}{=} (\mathrm{true}\,\mathtt{U}^{\alpha}\,\Phi)$ and

$$\exists[\![\alpha]\!]\Phi \overset{\mathrm{def}}{=} \neg\forall\langle\!\langle\alpha\rangle\!\rangle\neg\Phi \quad \text{and} \quad \forall[\![\alpha]\!]\Phi \overset{\mathrm{def}}{=} \neg\exists\langle\!\langle\alpha\rangle\!\rangle\neg\Phi.$$

The standard CTL operators for "next step", "until" and "eventually" are obtained by $\mathtt{X}\,\Phi \overset{\mathrm{def}}{=} (\mathrm{true}\,\mathtt{U}^{tt}\,\Phi) = \langle\!\langle tt\rangle\!\rangle\Phi$, $\Phi_1\,\mathtt{U}\,\Phi_2 \overset{\mathrm{def}}{=} (\Phi_1\,\mathtt{U}^{tt^*}\,\Phi_2)$ and $\Diamond\Phi \overset{\mathrm{def}}{=} (\mathrm{true}\,\mathtt{U}\,\Phi)$. The definition of the always operator $\Box$ in BTSL is as follows:

$$\exists\Box\Phi \overset{\mathrm{def}}{=} \neg\forall(\mathrm{true}\,\mathtt{U}\,\neg\Phi) \quad \text{and} \quad \forall\Box\Phi \overset{\mathrm{def}}{=} \neg\exists(\mathrm{true}\,\mathtt{U}\,\neg\Phi).$$

Other Boolean connectives, like disjunction or implication, are obtained in the obvious way.

Definition 13 (Semantics of BTSL)
Let $\mathcal{A} = (Q, \mathcal{N}, \mathcal{N}_{\mathrm{in}}, \mathcal{N}_{\mathrm{out}}, \longrightarrow, Q_0, \mathsf{AP}, L)$ be a labeled constraint automaton. The satisfaction relation $\models$ for BTSL state formulas is defined as follows:

$$\begin{array}{lll} q \models \mathrm{true} & & \\ q \models a & \iff & a \in L(q) \\ q \models \neg\Phi & \iff & q \not\models \Phi \\ q \models \Phi_1 \wedge \Phi_2 & \iff & q \models \Phi_1 \text{ and } q \models \Phi_2 \\ q \models \exists\varphi & \iff & \text{there exists a path } \pi \in \mathsf{Paths}(q) \text{ s.t. } \pi \models \varphi \\ q \models \forall\varphi & \iff & \text{for all paths } \pi \in \mathsf{Paths}(q)\text{: } \pi \models \varphi \end{array}$$

The meaning of a path formula is as follows. Let $\pi = q_0 \xrightarrow{c_1} q_1 \xrightarrow{c_2} \ldots$ be a path in $\mathcal{A}$. Then,

$$\pi \models \Phi_1 \, \mathsf{U}^{\alpha} \, \Phi_2 \ \text{ iff } \ \begin{array}{l} \text{there exists } n \in \mathbb{N} \text{ such that } \mathsf{ios}(\pi \downarrow n) \in \mathsf{IOS}(\alpha) \\ \text{and } q_i \models \Phi_1 \text{ for all } 0 \le i < n \text{ and } q_n \models \Phi_2 \end{array}$$

□

Let $\mathsf{Sat}(\Phi) = \{q \in Q \mid q \models \Phi\}$ be the satisfaction set of state formula Φ, consisting of all the states that satisfy Φ. A constraint automaton $\mathcal{A}$ then fulfills formula Φ, denoted as $\mathcal{A} \models \Phi$, if $q_0 \models \Phi$ for all initial states $q_0 \in Q_0$, i.e., $Q_0 \subseteq \mathsf{Sat}(\Phi)$.

Example 7. As an example we revisit the elevator system and provide some BTSL formulas. The formulas below can either be interpreted for the *Elevator* component in isolation (cf. Fig. 3) or for the composite system as a whole. The first formula Φ_1 states the existence of a path with a certain prefix which ends in a state where the elevator reached the third level.

$$\begin{aligned} \Phi_1 = \ & \exists\langle\!\langle(\{Level\} \wedge d_{Level} = 1)^*; \{Up\}; (\{Level\} \wedge d_{Level} = 2)^*; \\ & \{Up\}; (\{Level\} \wedge d_{Level} = 3)^*\rangle\!\rangle \texttt{"elevator at 3"} \end{aligned}$$

The I/O-stream of the path must be in the language of the given stream expression, stating that *Up* will be active exactly two times and there might be a finite number of requests for the current level information along this path.

The second formula Φ_2 states in essence that requesting the level information only, without performing any move, does not change the level of the elevator. Even more, we can specify that the fact that certain sequences of moves of the elevator eliminate the effect on its level. The formula Φ_3 formalizes this fact for sequences of arbitrary length consisting of either an *Up* followed by a *Down* or a *Down* followed by an *Up* move.

$$\Phi_2 = \bigwedge_{1 \le i \le k} \big(\texttt{"elevator at } i\texttt{"} \rightarrow \forall[\![\{Level\}^+]\!]\texttt{"elevator at } i\texttt{"}\big)$$

$$\begin{aligned} \Phi_3 = \bigwedge_{1 \le i \le k} \big(& \forall[\![\{step\}^*; (\{Level\} \wedge d_{Level} = i); \\ & ((\{Up\}; \{Down\}) \vee (\{Down\}; \{Up\}))^*]\!]\texttt{"elevator at } i\texttt{"}\big) \end{aligned}$$

The next formula Φ_4 states that incoming requests from level $1 \le i \le k$ to destination $1 \le j \le k$ have a chance of being answered eventually.

$$\begin{aligned} \Phi_4 = \bigwedge_{1 \le i,j \le k} \Big(& \forall[\![step^*; \{From, To\} \wedge d_{From} = i \wedge d_{To} = j]\!] \\ & \exists\Diamond\big(\texttt{"elevator at } i\texttt{"} \wedge \exists\Diamond(\texttt{"elevator at } j\texttt{"})\big)\Big) \end{aligned}$$

Model Checking BTSL Formulas. The model checking problem for BTSL asks whether, for a given constraint automaton $\mathcal{A}$ and BTSL state formula Φ, all initial states q_0 of $\mathcal{A}$ satisfy Φ. The main procedure for BTSL model checking follows the standard approach for CTL-like branching time logics [16] and recursively calculates the satisfaction sets

$$\mathsf{Sat}(\Psi) \overset{\text{def}}{=} \{q \in Q : q \models \Psi\}$$

for all subformulas Ψ of Φ. To compute the satisfaction sets of $\exists(\Phi_1 \,\mathtt{U}^{\alpha}\, \Phi_2)$ and $\forall(\Phi_1 \,\mathtt{U}^{\alpha}\, \Phi_2)$, we follow an automata-theoretic approach which resembles the standard automata-based LTL model checking procedure and relies on a representation of α by means of a finite automaton $\mathcal{Z}$ and the model checking of BTSL state formulas of the form $\exists(\Psi_1 \,\mathtt{U}\, \Psi_2)$ and $\forall(\Psi_1 \,\mathtt{U}\, \Psi_2)$, respectively, in the product of $\mathcal{A}$ and $\mathcal{Z}$. Using standard methods for regular languages, we first generate a finite automata $\mathcal{Z}$ over the alphabet CIO such that the accepted language of $\mathcal{Z}$ agrees with $\mathsf{IOS}(\alpha)$. In the sequel, let

$$\mathcal{Z} \;=\; (Z, \mathsf{CIO}, \longrightarrow_{\mathcal{Z}}, Z_0, Z_F),$$

where Z stands for the state space, Z_0 denotes the set of initial states, Z_F is the set of final (accept) states and $\longrightarrow_{\mathcal{Z}} \subseteq Z \times \mathsf{CIO} \times Z$ the transition relation. In fact, $\mathcal{Z}$ can be viewed as a constraint automaton where the set Z_F plays the role of the labeling function which separates the final states from the non-final states. Given $\mathcal{A}$ and $\mathcal{Z}$, we built the product $\mathcal{A} \bowtie \mathcal{Z}$, similar to the product of finite automata and the product operator for constraint automata.

Let $\mathcal{A}$ be a constraint automaton as in Def. 2 and $\mathcal{Z}$ an NFA as above. Furthermore, let Φ be a BTSL state formula. We define the constraint automaton $\mathcal{A} \bowtie_{\Phi} \mathcal{Z}$, or briefly $\mathcal{A} \bowtie \mathcal{Z}$ if Φ is clear from the context, as follows:

$$\mathcal{A} \bowtie \mathcal{Z} \overset{\text{def}}{=} (S, \mathcal{N}, \longrightarrow, S_0, \mathsf{AP}', L').$$

As the distinction between input and output data-flow locations is irrelevant at this level, we ignore $\mathcal{N}_{\text{in}}$ and $\mathcal{N}_{\text{out}}$. The state space S is $Q \times Z$, the set of initial states is given by $S_0 \overset{\text{def}}{=} \{\langle q_0, z_0\rangle \;:\; q_0 \in Q_0 \text{ and } z_0 \in Z_0\}$. The transitions in $\mathcal{A} \bowtie \mathcal{Z}$ are obtained by the following synchronization rule for concurrent I/O-operations $c \in \mathsf{CIO}$, states q in $\mathcal{A}$ and states $z \in Z$:

$$\frac{q \xrightarrow{N,g} q' \;\wedge\; c \in \mathsf{CIO}(N,g) \;\wedge\; z \xrightarrow{c}_{\mathcal{Z}} z'}{\langle q, z\rangle \xrightarrow{N,g_c} \langle q', z'\rangle}$$

where g_c is a data constraint enforcing the data assignments, i.e., such that $\mathsf{CIO}(N, g_c) = \{c\}$. The set of atomic propositions in $\mathcal{A} \bowtie \mathcal{Z}$ is $\mathsf{AP}' = \{a_{\Phi}, \mathsf{accept}\}$, while the labeling function L' is given by the requirements (i) $a_{\Phi} \in L'(\langle q, z\rangle)$ iff $q \models \Phi$ and (ii) $\mathsf{accept} \in L'(\langle q, z\rangle)$ iff $z \in Z_F$.

The following lemmas formalize the reduction of the model checking problem for BTSL state formulas of the form $\exists(\Phi_1 \,\mathtt{U}^{\alpha}\, \Phi_2)$ and $\forall(\Phi_1 \,\mathtt{U}^{\alpha}\, \Phi_2)$ to the problem of computing satisfaction sets for formulas of the type $\exists(\Psi_1 \,\mathtt{U}\, \Psi_2)$ and $\forall(\Psi_1 \,\mathtt{U}\, \Psi_2)$

in the product, respectively (see Fig. 13). For the treatment of formulas of the form $\forall(\Phi_1 \,\mathtt{U}^\alpha\, \Phi_2)$ we have to construct a deterministic finite automaton (DFA) $\mathcal{Z}$ for a stream expression α rather than a non-deterministic finite automaton (NFA) which may cause an exponential blowup in the size of $\mathcal{Z}$.

Lemma 1 (Treatment of $\exists(\Phi_1 \,\mathtt{U}^\alpha\, \Phi_2)$). *Let $\mathcal{A}$ be a constraint automaton, and $\mathcal{Z} = (Z, \mathsf{CIO}, \longrightarrow_{\mathcal{Z}}, Z_0, Z_F)$ an NFA for a stream expression α. Furthermore, let q be a state in $\mathcal{A}$, and Φ_1 and Φ_2 BTSL state formulas. Then, the following statements are equivalent:*

(a) $q \models \exists(\Phi_1 \,\mathtt{U}^\alpha\, \Phi_2)$ *in* $\mathcal{A}$
(b) $\langle q, z_0 \rangle \models \exists(a_{\Phi_1} \,\mathsf{U}\, (a_{\Phi_2} \wedge \mathsf{accept}))$ *in* $\mathcal{A} \bowtie \mathcal{Z}$ *for some* $z_0 \in Z_0$

Lemma 2 (Treatment of $\forall(\Phi_1 \,\mathtt{U}^\alpha\, \Phi_2)$). *Let $\mathcal{A}$, α, Φ_1, Φ_2 be as in Lemma 1 and $\mathcal{Z} = (Z, \mathsf{CIO}, \longrightarrow_{\mathcal{Z}}, z_0, Z_F)$ be a DFA for a stream expression α. Then, for all states $q \in Q$ the following statements are equivalent:*

(a) $q \models \forall(\Phi_1 \,\mathtt{U}^\alpha\, \Phi_2)$ *in* $\mathcal{A}$
(b) $\langle q, z_0 \rangle \models \forall(a_{\Phi_1} \,\mathsf{U}\, (a_{\Phi_2} \wedge \mathsf{accept}))$ *in* $\mathcal{A} \bowtie \mathcal{Z}$

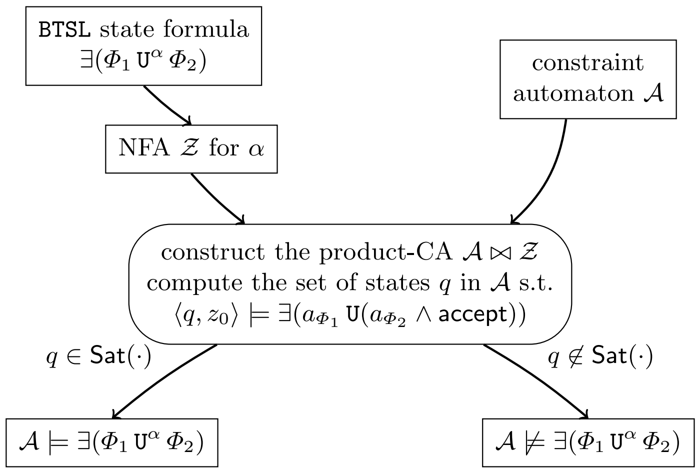

BTSL state formula $\exists(\Phi_1 \,\mathtt{U}^\alpha\, \Phi_2)$
constraint automaton $\mathcal{A}$
NFA $\mathcal{Z}$ for α
construct the product-CA $\mathcal{A} \bowtie \mathcal{Z}$ compute the set of states q in $\mathcal{A}$ s.t. $\langle q, z_0 \rangle \models \exists(a_{\Phi_1} \,\mathsf{U}(a_{\Phi_2} \wedge \mathsf{accept}))$
$q \in \mathsf{Sat}(\cdot)$
$q \notin \mathsf{Sat}(\cdot)$
$\mathcal{A} \models \exists(\Phi_1 \,\mathtt{U}^\alpha\, \Phi_2)$
$\mathcal{A} \not\models \exists(\Phi_1 \,\mathtt{U}^\alpha\, \Phi_2)$

Fig. 13. Schema for the treatment of $\exists(\Phi_1 \,\mathtt{U}^\alpha\, \Phi_2)$

Complexity. The complexity of BTSL model checking, i.e., computing the satisfaction sets of $\exists(\Phi_1 \,\mathtt{U}^\alpha\, \Phi_2)$ and $\forall(\Phi_1 \,\mathtt{U}^\alpha\, \Phi_2)$, is polynomial in the size of $\mathcal{A}$ and finite automaton $\mathcal{Z}$ for α. Thus, the overall time complexity of BTSL model checking for formulas of the form $\exists(\Phi_1 \,\mathtt{U}^\alpha\, \Phi_2)$ is polynomial in the size of $\mathcal{A}$ and the size of the input formula Φ, whereas the complexity for formulas of the form $\forall(\Phi_1 \,\mathtt{U}^\alpha\, \Phi_2)$ is polynomial in the size of $\mathcal{A}$ and exponential in the length of the input formula Φ due to the determinization of $\mathcal{Z}$.

4.4 Bisimulation Equivalence for Constraint Automata

The problem of checking bisimulation equivalence appears naturally in the design and optimization of complex systems. For example, given a complex component connector $\mathcal{C}$ that uses many internal channels, one might ask whether $\mathcal{C}$ can be replaced by a simpler connector $\mathcal{C}'$ that is cheaper according to some cost function. One possibility to verify that $\mathcal{C}$ and $\mathcal{C}'$ realize the same coordination mechanism is to prove the bisimulation equivalence of the constraint automata associated with $\mathcal{C}$ and $\mathcal{C}'$. Furthermore, bisimulation equivalence can also serve as a specification formalism. For example, the specification of a connector might be provided by means of a constraint automaton $\mathcal{A}_{\text{spec}}$ and the task is to provide the code for a connector $\mathcal{C}$ in Reo such that the constraint automata for $\mathcal{C}$ and $\mathcal{A}_{\text{spec}}$ are bisimulation equivalent.

Definition 14 (Bisimulation)
Let $\mathcal{A} = (Q, \mathcal{N}, \mathcal{N}_{\text{in}}, \mathcal{N}_{\text{out}}, \longrightarrow, Q_0, \mathsf{AP}, L)$ be a labeled constraint automaton. An equivalence relation $\mathcal{R}$ on Q is called bisimulation for $\mathcal{A}$ if for all pairs $(q_1, q_2) \in \mathcal{R}$ the following two conditions (i) and (ii) are satisfied:

(i) $L(q_1) = L(q_2)$
(ii) $\mathsf{CIO}(q_1, P) = \mathsf{CIO}(q_2, P)$ for all $\mathcal{R}$-equivalence classes $P \in Q/\mathcal{R}$

where $\mathsf{CIO}(q, P)$ for $q \in Q$ and $P \subseteq Q$ denotes the set of all I/O-operations that are enabled in state q and can lead to a state in P, i.e.,

$$\mathsf{CIO}(q, P) \;=\; \big\{ c \in \mathsf{CIO} \;:\; q \xrightarrow{N,g} p \text{ for some } p \in P \;\text{ and } c \in \mathsf{CIO}(N, g) \big\}.$$

Two states $q_1, q_2 \in Q$ are called bisimilar (or bisimulation equivalent) iff there exists a bisimulation $\mathcal{R}$ with $(q_1, q_2) \in \mathcal{R}$. □

As usual, the above definition of bisimulation equivalence for the states of a single constraint automaton can be adapted to define bisimulation equivalence of two constraint automata. Suppose that $\mathcal{A}_1$ and $\mathcal{A}_2$ are constraint automata with the same set of data-flow locations $\mathcal{N}$ and a common set AP of atomic propositions. Let $\mathcal{A}_1 \uplus \mathcal{A}_2$ be the "large" automaton obtained through the disjoint union of the state spaces of $\mathcal{A}_1$ and $\mathcal{A}_2$. Automata $\mathcal{A}_1$ and $\mathcal{A}_2$ are called bisimilar, denoted $\mathcal{A}_1 \sim \mathcal{A}_2$, if for each bisimulation equivalence class P in $\mathcal{A}_1 \uplus \mathcal{A}_2$ either P does not contain any initial state of $\mathcal{A}_1$ or $\mathcal{A}_2$ or P contains at least one initial state of both automata $\mathcal{A}_1$ and $\mathcal{A}_2$.

The classical partitioning refinement approach [25] for computing the bisimulation equivalence classes of a finite labeled transition system can be adapted for constraint automata [10,11]. This algorithm serves at the same time for checking bisimulation equivalence of two constraint automata and can also be used as a reduction technique by replacing a "large" constraint automaton with its the bisimulation quotient. Indeed the switch from a constraint automaton $\mathcal{A}$ to a bisimilar automaton $\mathcal{A}'$ preserves all properties that are expressible in the logics $\mathtt{LTL}_{\mathtt{IO}}$ and $\mathtt{BTSL}$.

Lemma 3. *If $\mathcal{A}_1 \sim \mathcal{A}_2$ then $\mathcal{A}_1$, $\mathcal{A}_2$ satisfy the same* BTSL *and* $\mathtt{LTL}_{IO}$ *formulas.*

The proof for these statements are standard (see e.g. [9]) and can be provided by structural induction. As for other CTL-like branching-time logics (see [13]), even a small fragment of BTSL is sufficient to provide a complete logical characterization of bisimulation equivalence. Constraint automata $\mathcal{A}_1$ and $\mathcal{A}_2$ are called equivalent with respect to a logic L, denoted $\mathcal{A}_1 \equiv_{\mathtt{L}} \mathcal{A}_2$, if $\mathcal{A}_1$ and $\mathcal{A}_2$ yield the same truth value for all formulas in L, i.e.,

$$\mathcal{A}_1 \equiv_L \mathcal{A}_2 \;\text{ iff }\; \text{for all } \phi \in \mathtt{L}\text{: } \mathcal{A}_1 \models \phi \iff \mathcal{A}_2 \models \phi$$

Let us now consider the sublogic L of BTSL consisting of all BTSL state formulas which can be build using the propositional fragment of BTSL (i.e., atomic propositions and the Boolean connectors $\wedge$ and $\neg$) and formulas of the form $\exists\langle\!\langle \mathsf{ioc}\rangle\!\rangle\Phi$ where Φ is a formula of L and ioc a basic stream expression given by an I/O-constraint (and representing a set consisting of I/O-streams of length 1). Then, $\mathcal{A}_1 \equiv_{\mathtt{L}} \mathcal{A}_2$ implies that $\mathcal{A}_1$ and $\mathcal{A}_2$ are bisimilar. Thus:

$$\mathcal{A}_1 \equiv_{\mathtt{L}} \mathcal{A}_2 \;\text{ iff }\; \mathcal{A}_1 \equiv_{\mathtt{BTSL}} \mathcal{A}_2 \;\text{ iff }\; \mathcal{A}_1 \sim \mathcal{A}_2$$

Hence, in order to show that two constraint automata are not bisimilar then a formula Φ in L can be provided that holds for $\mathcal{A}_1$, but not for $\mathcal{A}_2$. Such a formula Φ can be understood as a counterexample.

4.5 Vereofy

The Vereofy [12,8,7] tool-kit supports modeling and verification in the Reo and constraint automata framework. For modeling, it relies on a hybrid approach. Custom channels, connectors and the interface behavior of components can be modeled using CARML (Constraint Automata Reactive Module Language), a guarded command language for the concise modeling of constraint automata amenable to an efficient symbolic automaton representation. Reo networks for component connectors are built in RSL (Reo Scripting Language) by providing a script instantiating the various channels, connectors and components and plugging them together. Vereofy can be used as a stand-alone tool or integrated as a plugin in the Eclipse Coordination Tools [18], where Reo networks can be designed in a graphical way, allowing the visualization and animation of the data flow in the network, e.g., to investigate counter examples or witness generated during the model checking of a Reo network.

To cope with the state space explosion problem and typically large number of data-flow locations in a Reo network, Vereofy relies on a symbolic representation of the constraint automata. Vereofy provides model checking engines for properties specified in $\mathtt{LTL}_{\mathrm{IO}}$, BTSL and the alternating-time logic ASL [27] as well as for bisimulation checking. In the current implementation, Vereofy supports model checking the $\mathtt{LTL}_{\mathrm{IO}}$ fragment consisting of propositional logic, the standard LTL until operator (and the derived temporal operators) as well as the indexed next step operator $\langle\!\langle \mathsf{ioc}\rangle\!\rangle\varphi$ where ioc is an I/O-constraint. It also supports the use of enabled(ioc) to talk about the enabled concurrent I/O-operations at a state, which can be used to specify fairness conditions. The BTSL-fragment of our implementation cannot yet treat the $\mathtt{U}^{\alpha}$-operator, but directly supports the derived operators on path formulas $\langle\langle\alpha\rangle\rangle\Phi$ and $[\![\alpha]\!]\Phi$. The currently implemented version of the bisimulation algorithm abstracts away from state labels. Thus, it establishes equivalences only for the observable data flow.

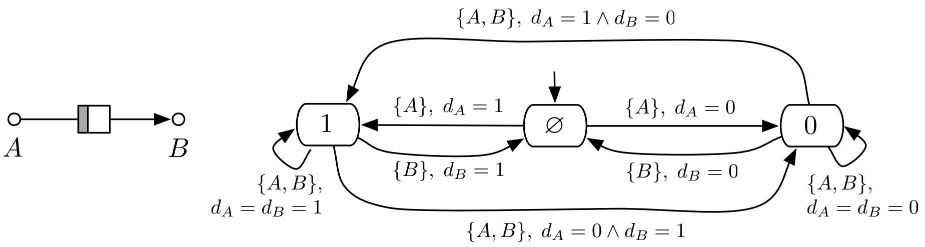
5 Realization of a Constraint Automaton by a Reo Network

We now address the question of how to realize a given constraint automaton $\mathcal{A}$ as a component connector. The motivation for this task originates from the classical (controller) synthesis problem, where the starting point is a formal model for an open system S (often called plant) and an objective Φ that formalizes the desired system properties and is typically given as a temporal formula. The task is then to design a controller C that restricts the possible behaviors of S (i.e., discards certain non-deterministic alternatives) such that the controlled system $S \parallel C$ meets the specification Φ. Several instances of the controller synthesis problem have been studied in the literature, e.g., [1,30,33,5,6] that differ in the type of system models and objectives, the assumptions on what is visible to the controller and the way how the environment and controller interact with S.

Here, we do assume that the system S is given as a constraint automaton $\mathcal{A}_S$ and that we have a specification of a controller as a constraint automaton $\mathcal{A}$ which ensures a certain property Φ, i.e., $\mathcal{A}_S \bowtie \mathcal{A} \models \Phi$. The goal is to synthesize the controller by a Reo network. In essential, the task is here to provide a construction of a Reo network $\mathcal{C}$ such that the constraint automaton $\mathcal{A}_{\mathcal{C}}$ for $\mathcal{C}$ is equal to $\mathcal{A}$ up to isomorphism. In [4] an algorithm has been presented that constructs a Reo network from a given constraint automaton. This approach is compositional and relies on a preprocessing step that generates an ω-regular expression from the given constraint automaton. We present here an alternative approach for the generation of a Reo network from the constraint automaton $\mathcal{A}$ which reuses some ideas of [4], but avoids the potential exponential blow-up in the construction of an ω-regular expression. We will first introduce the Reo primitives used as atomic building blocks in the construction.

Basic channels and component connectors. The construction makes use of synchronous channels and the synchronous drain channels as introduced in Section 3. Additionally, we will require a special variant of a FIFO1 channel where simultaneous writing and reading is possible if the buffer is filled. The effect of concurrent writing and reading is that the data item in the buffer is transmitted through the sink end and the new data item received at the source end is stored in the buffer. Fig. 14 shows the graphical representation of a *simultaneous FIFO1 channel* together with its constraint automaton.

In addition to the channels above the construction relies on use four component connectors that are used as "primitives":



A
B
$\{A,B\},\ d_A = 1 \wedge d_B = 0$
$\{A\},\ d_A = 1$
$\{A\},\ d_A = 0$
1
∅
0
$\{B\},\ d_B = 1$
$\{B\},\ d_B = 0$
$\{A,B\},$ $d_A = d_B = 1$
$\{A,B\},$ $d_A = d_B = 0$
$\{A,B\},\ d_A = 0 \wedge d_B = 1$

Fig. 14. Simultaneous FIFO1 channel for data domain $\mathsf{Data} = \{0, 1\}$

1. A *merger* has several input ports $A_1, \ldots, A_r$ and one output port B. It accepts non-deterministically data from exactly one of the input ports and forwards it synchronously through the output port.
2. An *exclusive router* has one input port A and several output ports $B_1, \ldots, B_r$ synchronously routes an incoming data from port A to exactly one of its output ports.
3. A *replicator* has one input port A and several output ports $B_1, \ldots, B_r$. It sends copies of an incoming data to all of its output ports synchronously.
4. A *data constraint checker* for a data constraint

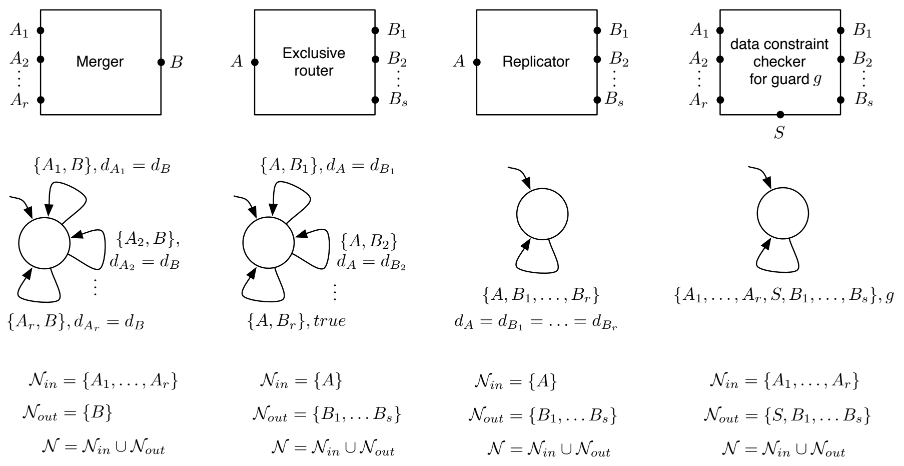
$$g \in DC(A_1, \ldots, A_r, B_1, \ldots, B_s)$$

has input ports $A_1, \ldots, A_r$ and output ports $S, B_1, \ldots, B_s$. Both, the input and output ports must be active synchronously and the observed data must fulfill the data constraint g. The additional output port S synchronously fires a token to indicate that the data has been accepted.

Fig. 15 shows the graphical representation of these four component connectors together with their constraint automata.

Remark 2. We will treat the constraint checkers as primitives, but using the approach presented in [4] the constraint checkers themselves could also be realized by a Reo network. For data constraints in a canonical (disjunctive) normal form the idea is to provide simple Reo networks for the literals $d_A = c$, $d_A = d_B$, etc. and component connectors that realize conjunctions and disjunctions. In the synthesis algorithm proposed in this section we will use standard Reo nodes with the standard and routing semantics instead of mergers, exclusive routers, and replicators. The standard Reo node behavior corresponds to the product of a merger and a replicator (cf. conditions (S1) and (S2) in Sec. 3), whereas the routing behavior corresponds to the composition of a merger and an exclusive router (cf. conditions (R1) and (R2) in Sec. 3).

Assumptions. In the sequel, we suppose that we are given a constraint automaton $\mathcal{A} = (Q, \mathcal{N}, \mathcal{N}_{\text{in}}, \mathcal{N}_{\text{out}}, \longrightarrow, \{q_0\})$. To simplify the presentation in this paper we will present the construction for constraint automata with a single



A_1
A_2
A_r
Merger
B
A
Exclusive router
B_1
B_2
B_s
A
Replicator
B_1
B_2
B_s
A_1
A_2
A_r
data constraint checker for guard g
B_1
B_2
B_s
S
$\{A_1, B\}, d_{A_1} = d_B$
$\{A_2, B\}, d_{A_2} = d_B$
$\{A_r, B\}, d_{A_r} = d_B$
$\{A, B_1\}, d_A = d_{B_1}$
$\{A, B_2\}$ $d_A = d_{B_2}$
$\{A, B_r\}, true$
$\{A, B_1, \ldots, B_r\}$ $d_A = d_{B_1} = \ldots = d_{B_r}$
$\{A_1, \ldots, A_r, S, B_1, \ldots, B_s\}, g$
$\mathcal{N}_{in} = \{A_1, \ldots, A_r\}$ $\mathcal{N}_{out} = \{B\}$ $\mathcal{N} = \mathcal{N}_{in} \cup \mathcal{N}_{out}$
$\mathcal{N}_{in} = \{A\}$ $\mathcal{N}_{out} = \{B_1, \ldots B_s\}$ $\mathcal{N} = \mathcal{N}_{in} \cup \mathcal{N}_{out}$
$\mathcal{N}_{in} = \{A\}$ $\mathcal{N}_{out} = \{B_1, \ldots B_s\}$ $\mathcal{N} = \mathcal{N}_{in} \cup \mathcal{N}_{out}$
$\mathcal{N}_{in} = \{A_1, \ldots, A_r\}$ $\mathcal{N}_{out} = \{S, B_1, \ldots B_s\}$ $\mathcal{N} = \mathcal{N}_{in} \cup \mathcal{N}_{out}$

Fig. 15. Merger, exclusive router, replicator and data constraint checker

starting state q_0. The treatment of multiple possible starting states requires for an additional component which non-deterministically selects the initial state and transition that fires first. Furthermore, we assume that all data-flow locations $A \in \mathcal{N}$ are either contained in $\mathcal{N}_{\text{in}}$ or in $\mathcal{N}_{\text{out}}$, i.e., $\mathcal{N} = \mathcal{N}_{\text{in}} \cup \mathcal{N}_{\text{out}}$.

Idea of the construction. The idea of creating a Reo network $\mathcal{C}$ that realizes $\mathcal{A}$ is to represent each state $q \in Q$ by a simultaneous FIFO1 channel f_q with a single buffer cell. The Reo network will mimic $\mathcal{A}$'s behavior by a *token game*, where exactly one of these simultaneous FIFO1 channels will be filled at a time. Initially, the token is in the buffer of f_{q_0}. Whenever a transition $q \xrightarrow{N,g}_{\mathcal{A}} q'$ fires the token moves from the buffer of f_q to the buffer of $f_{q'}$. The structure of the construction is shown in Fig. 16. For state q in $\mathcal{A}$, we deal with the simultaneous FIFO1 channel f_q and an exclusive router EXR_q and a merger MGR_q. The exclusive router EXR_q has one output port for each transition emanating in q. The merger MGR_q has one input port for each transition ending in q. For each transition θ in $\mathcal{A}$ there is a replicator REP_θ and a data constraint checker DCC_θ. Furthermore, for each output port $A \in \mathcal{N}_{\text{out}}$ the Reo network contains a merger MGR_A. Dually, for each input port $B \in \mathcal{N}_{\text{in}}$ we deal with an exclusive router EXR_B. Their interface ports are connected with the data constraint checkers of the transitions the ports are involved in.

Each of the simultaneous FIFO1 channels f_q is connected to the corresponding exclusive router EXR_q. The exclusive router "schedules" non-deterministically one of the outgoing transitions θ and routes the token through the corresponding output port into the replicator REP_θ. The first task of this replicator is to forward the token towards the simultaneous FIFO $f_{q'}$. The merger $\text{MGR}_{q'}$ ensures that only one of the incoming transitions of state q' can fire at a time. The second task of the replicator REP_θ is to synchronize the transition with the

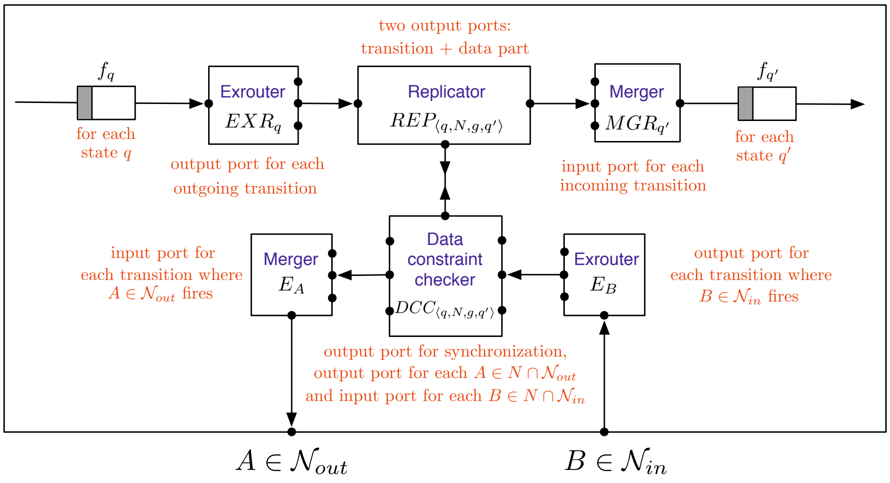

two output ports:
transition + data part
f_q
for each state q
Exrouter
EXR_q
output port for each outgoing transition
Replicator
$REP_{\langle q,N,g,q'\rangle}$
Merger
$MGR_{q'}$
input port for each incoming transition
$f_{q'}$
for each state q'
input port for each transition where $A \in \mathcal{N}_{out}$ fires
Merger
E_A
Data constraint checker
$DCC_{\langle q,N,g,q'\rangle}$
Exrouter
E_B
output port for each transition where $B \in \mathcal{N}_{in}$ fires
output port for synchronization, output port for each $A \in N \cap \mathcal{N}_{out}$ and input port for each $B \in N \cap \mathcal{N}_{in}$
$A \in \mathcal{N}_{out}$
$B \in \mathcal{N}_{in}$

Fig. 16. Structure of a Reo network $\mathcal{C}$ synthesized from constraint automaton $\mathcal{A}$

data constraint checker DCC_θ. Thus, the transition $\theta = (q, N, g, q')$ can fire if and only if all ports $A \in N$ fire and the observed data fulfills the data constraint $g \in DC(N)$. For each of the ports $A \in \mathcal{N}$ we have to ensure that they are not involved in more than one transition at a time.

For this purpose, we make use of the merger components E_A in case of the output ports $\mathcal{A} \in \mathcal{N}_{\text{out}}$ and the exclusive routers E_B for the input ports $B \in \mathcal{N}_{\text{in}}$ of $\mathcal{A}$. The synchronous channels from the output port of E_A to A and from the data constraint checkers DCC_θ to E_A ensure that data flow at A is always synchronized with data flow at the data constraint checker DCC_θ and the replicator REP_θ of some transition $\theta = (q, N, g, q')$ if $A \in N$. Similarly, the synchronous channels from B to input port of E_B and from E_B to the data constraint checkers DCC_θ ensure that data flow at B is always synchronized with data flow at DCC_θ and REP_θ of some transition $\theta = (q, N, g, q')$ if $B \in N$ Vice versa, whenever there is some data flow at one of the replicators REP_θ then there must be data flow at the corresponding data constraint checker DCC_θ and all active ports, i.e., those that appear in the set N of θ.

Soundness of the construction. Let $\mathcal{A} = (Q_\mathcal{A}, \mathcal{N}_\mathcal{A}, \mathcal{N}^\mathcal{A}_{\text{in}}, \mathcal{N}^\mathcal{A}_{\text{out}}, \longrightarrow_\mathcal{A}, \{q_0\})$ be a constraint automaton and $\mathcal{C}$ be the Reo network synthesized from $\mathcal{A}$. Using the constraint automata product as presented in Section 3.1 we can compose an automaton representation for $\mathcal{C}$. Let $\mathcal{A}_\mathcal{C} = (Q_\mathcal{C}, \mathcal{N}_\mathcal{C}, \mathcal{N}^\mathcal{C}_{\text{in}}, \mathcal{N}^\mathcal{C}_{\text{out}}, \longrightarrow_\mathcal{C}, Q_{0,\mathcal{C}})$ denote the constraint automaton resulting from the the product of all component connectors, channels and nodes of $\mathcal{C}$. The set $\mathcal{N}_\mathcal{C}$ contains all internal nodes and boundary nodes (i.e., ports) of $\mathcal{C}$, while $\mathcal{N}^\mathcal{C}_{\text{in}} = \mathcal{N}^\mathcal{A}_{\text{in}}$ and $\mathcal{N}^\mathcal{C}_{\text{out}} = \mathcal{N}^\mathcal{A}_{\text{out}}$.

Lemma 4 (Soundness of the construction). *Let $\mathcal{A}$ be a constraint automaton and let $\mathcal{C}$ be the constructed Reo network with constraint automaton product*

$\mathcal{A}_\mathcal{C}$ as above. Then, the Reo network $\mathcal{C}$ correctly implements $\mathcal{A}$, i.e., the reachable fragments of the constraint automata $\mathcal{A}_\mathcal{C}$ and $\mathcal{A}$ are isomorphic.

Proof sketch. The state space of the constraint automaton $\mathcal{A}_\mathcal{C}$ is the Cartesian product of the states of the FIFO1 channels f_q that store the token in $\mathcal{C}$, i.e., $Q_\mathcal{C} = \{\mathtt{empty}, \mathtt{full}\}^n$, where $n = |Q|$ is the number of states in the original constraint automaton $\mathcal{A}$. The token game starts with a single FIFO1 channel being full and passes the token to exactly one other FIFO1 channel in each step. We denote by $Q'_\mathcal{C} \subseteq Q_\mathcal{C}$ the states of $\mathcal{A}_\mathcal{C}$ where exactly one FIFO1 channel is full, i.e., the states that are relevant for the token game. By construction, the reachable fragment of $\mathcal{A}_\mathcal{C}$, i.e., the states that can be reached via a finite execution from an initial state, is contained in $Q'_\mathcal{C}$. To relate the states of $\mathcal{A}$ and $\mathcal{A}_\mathcal{C}$, let $h : Q_\mathcal{A} \to Q'_\mathcal{C}$ be the bijection that maps each state $q \in Q_\mathcal{A}$ to the corresponding state $h(q)$, i.e., the state of $\mathcal{A}_\mathcal{C}$ where f_q is $\mathtt{full}$ while all the other buffers $f_{q'}$ with $q' \neq q$ are $\mathtt{empty}$.

Furthermore, we have for all $q, q' \in Q_\mathcal{A}$, $N \subseteq \mathcal{N}_\mathcal{C}$ and $g \in DC(N)$:

$$h(q) \xrightarrow{N,g}_{\mathcal{A}_\mathcal{C}} h(q') \qquad \text{iff} \qquad q \xrightarrow{N',g'}_{\mathcal{A}} q' \tag{1}$$

where $N' = N \cap \mathcal{N}_\mathcal{A}$ and $g' \equiv \exists[\mathcal{N}_\mathcal{C} \backslash \mathcal{N}_\mathcal{A}]g$, i.e., where N', g' corresponds to N, g after all the internal nodes A in $\mathcal{C}$, $A \in \mathcal{N}_\mathcal{C} \setminus \mathcal{N}_\mathcal{A}$, have been hidden.

To verify equation (1), one has to apply the product construction for constraint automata (cf. Def. 5) to all channels and component connectors (exclusive routers, replicators, mergers, and constraint checkers) that appear in the Reo network $\mathcal{C}$. We conclude that, for a state q in $\mathcal{A}$ and the corresponding state $h(q)$ in $\mathcal{A}_\mathcal{C}$, the same concurrent I/O-operations are enabled in q and $h(q)$ after hiding the internals of $\mathcal{A}_\mathcal{C}$. Thus the reachable fragments of the automaton $\mathcal{A}_\mathcal{C}$ for the Reo network $\mathcal{C}$ and the constraint automaton $\mathcal{A}$ are isomorphic after "hiding" all internals of $\mathcal{C}$, i.e., after applying the hide operator for constraint automata to $\mathcal{A}_\mathcal{C}$ to hide the nodes $\mathcal{N}_\mathcal{C} \setminus \mathcal{N}_\mathcal{A}$ only occurring in $\mathcal{A}_\mathcal{C}$.

□

Example 8 (Synthesis). We close this section on the synthesis of a Reo network from a constraint automaton by an example. Our starting point is the constraint automaton $\mathcal{A} = (Q, \mathcal{N}, \mathcal{N}_{\text{in}}, \mathcal{N}_{\text{out}}, \longrightarrow, \{q_0\})$ with state space $Q = \{q_0, q_1, q_2\}$, the set of data-flow locations $\mathcal{N} = \{A_1, A_2, A_3, B_1, B_2, B_3\}$, and transition relation as depicted in Fig. 17. The data-flow locations are disjointly partitioned into the set of output ports $\mathcal{N}_{\text{out}} = \{A_1, A_2, A_3\}$ and the set of input ports $\mathcal{N}_{\text{in}} = \{B_1, B_2, B_3\}$.

The constructed Reo network $\mathcal{C}$ for $\mathcal{A}$ is shown in Fig. 18. As explained in Remark 2 the replicators suggested in Fig. 16 have been replaced by standard replicating Reo nodes and the exclusive routers by routing Reo nodes as their behavior agrees with the corresponding component connector. □

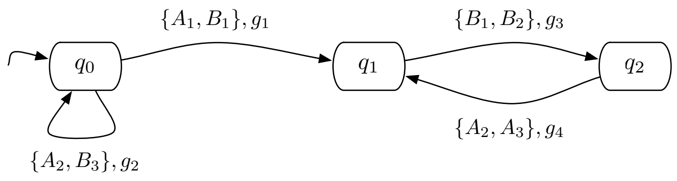


Fig. 17. Example of a constraint automaton to be synthesized

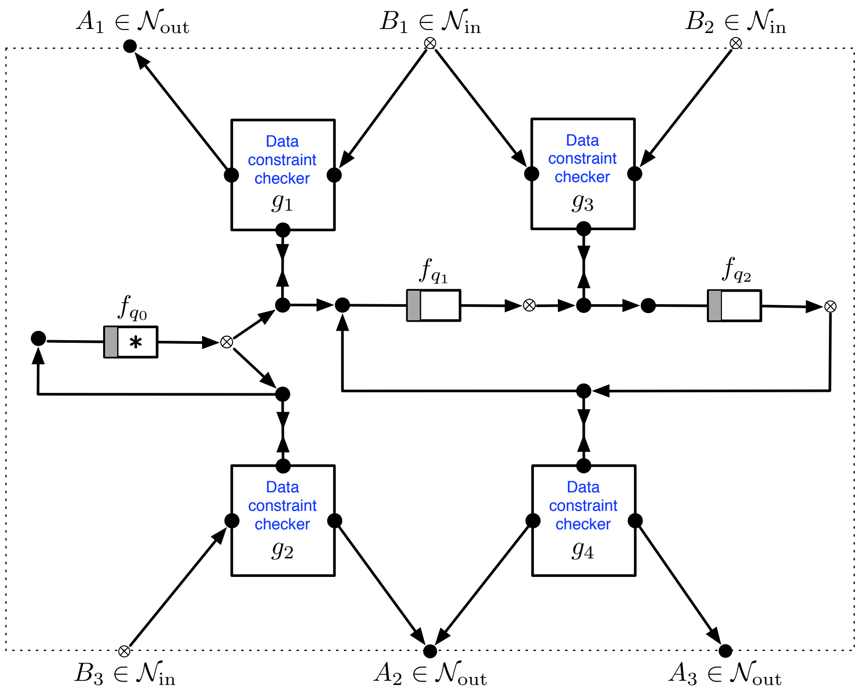

$A_1 \in \mathcal{N}_{\text{out}}$
$B_1 \in \mathcal{N}_{\text{in}}$
$B_2 \in \mathcal{N}_{\text{in}}$
Data constraint checker g_1
Data constraint checker g_3
f_{q_0}
f_{q_1}
f_{q_2}
Data constraint checker g_2
Data constraint checker g_4
$B_3 \in \mathcal{N}_{\text{in}}$
$A_2 \in \mathcal{N}_{\text{out}}$
$A_3 \in \mathcal{N}_{\text{out}}$

Fig. 18. Synthesized Reo network $\mathcal{C}$ for the automaton $\mathcal{A}$ of Fig. 17

6 Conclusion

We presented an overview of the basic concepts of the modeling and formal verification of models specified in the Reo and constraint automaton framework. Constructing the coordination glue code from basic coordination primitives like channels and component connectors provides an intuitive, hierarchical approach for modeling a wide variety of coordination and communication mechanisms. The hybrid modeling approach of either specifying behavior via a Reo network or directly as a constraint automaton allows the modeling at the appropriate level of abstraction, e.g., a component may be in a first iteration specified by a constraint automaton describing its interface behavior in an abstract fashion and then later replaced by a refined version realized by a Reo network. The logics and verification algorithms presented in this paper support the specification and automatic verification of a wide variety of relevant properties. The Vereofy tool-kit, implementing the modeling and verification approaches outlined in this paper, has been successfully used in the modeling and verification of a number of academic examples as well as two industrial case studies in the context of wireless sensor networks [14] and of a distributed telephony platform.

References

1. Abadi, M., Lamport, L., Wolper, P.: Realizable and unrealizable specifications of reactive systems. In: Ronchi Della Rocca, S., Ausiello, G., Dezani-Ciancaglini, M. (eds.) ICALP 1989. LNCS, vol. 372, pp. 1–17. Springer, Heidelberg (1989)
2. Arbab, F.: Reo: A Channel-Based Coordination Model for Component Composition. Mathematical Structures in Computer Science 14(3), 329–366 (2004)
3. Arbab, F., Baier, C., de Boer, F., Rutten, J.: Models and temporal logical specifications for timed component connectors. Software and System Modeling 6(1), 59–82 (2007)
4. Arbab, F., Baier, C., de Boer, F., Rutten, J., Sirjani, M.: Synthesis of Reo Circuits for Implementation of Component-Connector Automata Specifications. In: Jacquet, J.-M., Picco, G.P. (eds.) COORDINATION 2005. LNCS, vol. 3454, pp. 236–251. Springer, Heidelberg (2005)
5. Asarin, E., Bournez, O., Dang, T., Maler, O., Pnueli, A.: Effective Synthesis of Switching Controllers for Linear Systems. IEEE Special Issue on Hybrid Systems 88, 1011–1025 (2000)
6. Asarin, E., Maler, O., Pnueli, A.: Symbolic Controller Synthesis for Discrete and Timed Systems. In: Antsaklis, P.J., Kohn, W., Nerode, A., Sastry, S.S. (eds.) HS 1994. LNCS, vol. 999, pp. 1–20. Springer, Heidelberg (1995)
7. Baier, C., Blechmann, T., Klein, J., Klüppelholz, S.: A Uniform Framework for Modeling and Verifying Components and Connectors. In: Field, J., Vasconcelos, V.T. (eds.) COORDINATION 2009. LNCS, vol. 5521, pp. 247–267. Springer, Heidelberg (2009)
8. Baier, C., Blechmann, T., Klein, J., Klüppelholz, S.: Formal Verification for Components and Connectors. In: de Boer, F.S., Bonsangue, M.M., Madelaine, E. (eds.) FMCO 2008. LNCS, vol. 5751, pp. 82–101. Springer, Heidelberg (2009)
9. Baier, C., Katoen, J.-P.: Principles of Model Checking. The MIT Press, Cambridge (2008)
10. Baier, C., Sirjani, M., Arbab, F., Rutten, J.: Modeling Component Connectors in Reo by Constraint Automata. Science of Computer Programming 61(2), 75–113 (2006)
11. Blechmann, T., Baier, C.: Checking equivalence for Reo networks. In: FACS 2007. Electronic Notes in Theoretical Computer Science, vol. 215, pp. 209–226. Elsevier Publishers B.V., Amsterdam (2008)
12. Blechmann, T., Klein, J., Klüppelholz, S.: Vereofy User Manual. Technische Universität Dresden (2008–2011), `http://www.vereofy.de/`
13. Browne, M., Clarke, E., Grumberg, O.: Characterizing Finite Kripke Structures in Propositional Temporal Logic. Theoretical Computer Science 59(1-2), 115–131 (1988)

14. Baier, C., Blechmann, T., Klein, J., Klüppelholz, S., Leister, W.: Design and Verification of Systems with Exogenous Coordination Using Vereofy. In: Margaria, T., Steffen, B. (eds.) ISoLA 2010. LNCS, vol. 6416, pp. 97–111. Springer, Heidelberg (2010)
15. Clarke, D., Costa, D., Arbab, F.: Modelling Coordination in Biological Systems. In: Margaria, T., Steffen, B. (eds.) ISoLA 2004. LNCS, vol. 4313, pp. 9–25. Springer, Heidelberg (2006)
16. Clarke, E., Emerson, E., Sistla, A.: Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. ACM Transactions on Programming Languages and Systems 8(2), 244–263 (1986)
17. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (1999)
18. Eclipse Coordination Tools, `http://reo.project.cwi.nl/`
19. Fischer, M., Ladner, R.: Propositional Dynamic Logic of Regular Programs. Journal of Computer and System Science 8, 194–211 (1979)
20. Francez, N.: Fairness. Texts and Monographs in Computer Science. Springer, Heidelberg (1986)
21. Giordano, L., Martelli, A.: Tableau-based automata construction for dynamic linear time temporal logic. Annals of Mathematics and Artificial Intelligence 46(3), 289–315 (2006)
22. Henriksen, J., Thiagarajan, P.: Dynamic Linear Time Temporal Logic. Annals of Pure and Applied Logic 96(1-3), 187–207 (1999)
23. Hoare, C.: Communcating Sequential Processes. Prentice-Hall, Englewood Cliffs (1985)
24. Holzmann, G.: Design and Validation of Computer Protocols. Prentice-Hall, Englewood Cliffs (1990)
25. Kanellakis, P., Smolka, S.: CCS Expressions, Finite State Processes, and Three Problems of Equivalence. Information and Computation 86(1), 43–68 (1990)
26. Klüppelholz, S., Baier, C.: Symbolic model checking for channel-based component connectors. Science of Computer Programming 74(9), 688–701 (2009)
27. Klüppelholz, S., Baier, C.: Alternating-time stream logic for multi-agent systems. Science of Computer Programming 75(6), 398–425 (2010)
28. Milner, R.: Communication and Concurrency. International Series in Computer Science. Prentice-Hall, Englewood Cliffs (1989)
29. Pnueli, A.: The Temporal Logic of Programs. In: Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science, pp. 46–57. IEEE Computer Society Press, Los Alamitos (1977)
30. Pnueli, A., Rosner, R.: On the Synthesis of a Reactive Module. In: Proceedings of the 16th Annual ACM Symposium on Principles of Programming Languages, pp. 179–190. ACM Press, New York (1989)
31. Vardi, M.: An Automata-Theoretic Approach to Linear Temporal Logic. In: Moller, F., Birtwistle, G. (eds.) Logics for Concurrency. LNCS, vol. 1043, pp. 238–266. Springer, Heidelberg (1996)
32. Vardi, M.Y., Wolper, P.: An Automata-Theoretic Approach to Automatic Program Verification. In: Proceedings of the 1st Annual Symposium on Logic in Computer Science, pp. 332–345. IEEE Computer Society Press, Los Alamitos (1986)
33. Wonham, W.: On the control of discrete-event systems. In: Three Decades of Mathematical System Theory. Lecture Notes in Control and Information Sciences, vol. 135, pp. 542–562. Springer, Heidelberg (1989)

Application-Layer Connector Synthesis[⋆]

Paola Inverardi, Romina Spalazzese, and Massimo Tivoli

Dipartimento di Informatica - Università degli Studi dell'Aquila, Italy
{paola.inverardi,romina.spalazzese,massimo.tivoli}@di.univaq.it

Abstract. The heterogeneity characterizing the systems populating the Ubiquitous Computing environment prevents their seamless interoperability. Heterogeneous protocols may be willing to cooperate in order to reach some common goal even though they meet dynamically and do not have a priori knowledge of each other. Despite numerous efforts have been done in the literature, the automated and run-time interoperability is still an open challenge for such environment. We consider interoperability as the ability for two Networked Systems (NSs) to communicate and correctly coordinate to achieve their goal(s).

In this chapter we report the main outcomes of our past and recent research on automatically achieving protocol interoperability via connector synthesis. We consider *application-layer connectors* by referring to two conceptually distinct notions of connector: *coordinator* and *mediator*. The former is used when the NSs to be connected are already able to communicate but they need to be specifically coordinated in order to reach their goal(s). The latter goes a step forward representing a solution for both achieving correct coordination and enabling communication between highly heterogeneous NSs. In the past, most of the works in the literature described efforts to the automatic synthesis of coordinators while, in recent years the focus moved also to the automatic synthesis of mediators. Within the Connect project, by considering our past experience on automatic coordinator synthesis as a baseline, we propose a formal theory of mediators and a related method for automatically eliciting a way for the protocols to interoperate. The solution we propose is the *automated synthesis of emerging mediating connectors* (i.e., *mediators* for short).

1 Introduction

Today's ubiquitous computing environment is populated by a wide variety of heterogeneous Networked Systems (NSs), dynamically appearing and disappearing, that belong to a multitude of application domains: home automation, consumer electronics, mobile and personal computing, to mention a few. Key technologies such as the Internet, the Web, and the wireless computing devices and networks can be qualified as ubiquitous, in the sense of Mark Weiser [80], even if these technologies have still not reached the maturity envisioned by the Ubiquitous Computing and the subsequent pervasive computing and ambient intelligence

[⋆] This work has been partially supported by the FET project Connect No 231167.

paradigms because of the extreme level of heterogeneity of the underlying infrastructure which prevents seamless interoperability. In this environment, heterogeneous protocols may be willing to cooperate in order to reach some common goal even though they meet dynamically and do not have a priori knowledge of each other.

The term *protocol* refers to *interaction protocols* or *observable protocols*. That is, a protocol is the behavior of a system in terms of the sequences of messages visible at the interface level, which it exchanges with other systems. In this chapter we consider *application-layer* protocols as opposed to *midlleware-layer* protocols that are treated in detail in [76].

By referring to the notion of *interoperability* introduced in [30], the problem we address in this chapter, is related to *how to automatically achieve the interoperability between heterogeneous protocols in the Ubiquitous Computing environment.*

With interoperability, we mean the ability of heterogeneous protocols to communicate and correctly coordinate to achieve their goal(s). The communication is expressed as synchronization, i.e., two systems communicate if they are able to synchronize on “common actions”. Coordination is expressed by the achievement of a specified goal, i.e., two systems succeed in coordinating if they interact through synchronization according to the achievement of their goal(s). Communication that is achieved through a complex protocols interaction can be regarded as a simple form of coordination. Indeed, application level protocols introduce a notion of communication that goes beyond single basic synchronizations and may require a well defined sequence of synchronization to be achieved.

In order to make communication and correct (with respect to the specified goal) coordination between heterogeneous protocols possible, we focus on methods, and related tools, for the *automatic application-layer connector synthesis*. In particular, in this chapter, we report our past and recent work on devising automatic connector synthesis techniques in the domains of *Component Based Software Engineering* (CBSE) and *Ubiquitous Computing* (UbiComp), respectively. The work carried on within the CBSE domain can be considered as a baseline for the work done in the UbiComp domain. The latter has been done in the context of the CONNECT project [30] and, with respect to our past work, represents the novel contribution concerning the automatic synthesis of application-layer connectors. However, it is worth mentioning that these two research contributions address two distinct sub-problems of the automatic connector synthesis problem.

In particular, in the CBSE domain, we used automatic connector synthesis in order to face the so-called *component assembly* problem. This problem can be considered as a particular instance of the above mentioned interoperability problem where the issue of enabling communication is assumed to be (almost) already solved. The focus, in the component assembly problem, is on how to coordinate the interactions of already communicating black-box components so that the resulting system is free from possible deadlocks and it satisfies a goal specified in terms of *coordination policies*. Dealing with black-box components, this is done by inserting in the system a software *coordinator*. It is an additional component beyond the ones forming the system and it is synthesized so as to intercept all component interactions in order to prevent deadlocks and those interactions that violate the specified coordination policies. Coordination policies are routing policies usually specified in some automata-based or temporal logic formalism. Thus, a *coordinator* can be considered as a specific notion of connector, i.e., a *coordination connector*.

Conversely, in the UbiComp domain, the granularity of a system shifts from the granularity of a system of components (as in the CBSE domain) to the one of a *System-of-Systems* (SoS) [27]. An SoS is characterized by an assembly of a wide variety of building blocks. Thus, in the UbiComp domain, enabling communication between heterogeneous NSs regardless, at a first stage, possible coordination mismatches, becomes a primary concern. This introduces another specific notion of connector, i.e., the notion of *mediator* seen as a *communication connector*.

Achieving correct communication and coordination among heterogeneous NSs means achieving interoperability among them. The interoperability problem and the specific notions of connector (e.g., coordinator or mediator) that can be used to solve it, or part of it, have been the focus of extensive studies within different research communities. Protocol interoperability come from the early days of networking and different efforts, both theoretical and practical, have been done to address it in several areas including, for example: protocol conversion, component adaptors, Web services mediation, theories of connectors, wrappers, bridges, and interoperability platforms.

Despite the existence of numerous solutions in the literature, to the best of our knowledge, all of them are more focused on coordinator synthesis and little effort has been devoted to the automatic synthesis of mediators. In particular, these approaches either: (i) assume the communication problem solved (or almost solved) by considering protocols already (or almost) able to interoperate; or (ii) are informal making automatic reasoning impossible; or (iii) follows a semi-automatic process for the mediator synthesis requiring a human intervention; or (iv) consider only few possible mismatches.

Our recent work on mediator synthesis has been devoted in particular to (i) the elicitation and definition of a theory of emerging connectors which also includes related supporting methods and tools. In particular, our recent work has led us *to design automated model-based techniques and tools to support the devised synthesis process*, from protocol abstraction to matching and mapping. Moreover we (ii) characterized protocol mismatches and related mediator patterns, and (iii) we designed a combined approach to take into consideration also non-functional properties while building an interoperability solution. While (i) is part of this chapter, for (ii) and (iii), we refer to [66,65] and [14], respectively.

The remainder of the chapter is organized as follows. Section 2 sets the context of the work reported in this chapter. In particular, by means of two examples, this section clarifies the distinction between the notions of coordinator and mediator. Section 3 describes different approaches for the automatic synthesis of coordinators. Section 4 describes the theory of emerging connectors (i.e., of mediators) mentioned above. Since the pool of coordinator synthesis approaches that are discussed in Section 3 represents the *baseline* chosen from the state-of-the-art in order to devise the approach described in Section 4, for the sake of brevity, the level of description of these two sections is intentionally kept different. That is, Section 3 briefly recalls the different coordinator synthesis approaches by simply providing an overview of them, whereas Section 4 describes in more detail the novel contribution of our recent research with respect to the automatic synthesis of application-layer connectors. Section 5 discusses related works in the areas of both coordinator and mediator synthesis. Section 6 concludes the chapter and outlines our future perspectives in the context of Connect.

2 Setting the Context

Within the Connect project, at synthesis stage, we can assume that a NS comes together with a *Labeled Transition System* (LTS) [42] based specification of its interaction protocol. The interaction protocol of a NS expresses the order in which input and output actions are performed while the NS interacts with its environment. Input actions model methods that can be called, or the end of receiving messages from communication channels, as well as the return values from such calls. Output actions model method calls, message transmission via communication channels, or exceptions that occur during methods execution.

As said in Section 1, our focus is on the automatic synthesis of *application-layer connectors*. Our notion of protocol abstracts from the content of the exchanged data, i.e., values of method/operation parameters, return values, or content of messages. That is, we are interested in harmonizing the behavior protocol (e.g., scheduling of operation calls) of heterogeneous NSs rather than performing mediation of communication primitives or of data encoding/decoding that are issues related to the synthesis of middleware-layer connectors (see the work described in [76]).

As introduced in Section 1, the interoperability problem concerns the problem of both enabling *communication* and achieving correct *coordination*. We recall that in our past research we addressed correct coordination by assuming communication already solved. This is done via automatic coordinator synthesis (Section 3). Instead, our current research focuses on the whole interoperability problem by devising methods and tools for the automatic mediator synthesis (Section 4).

In order to better clarify the distinction between the notions of coordinator and mediator, in the following two sub-sections, we describe two simple yet significant examples of the kinds of interoperability problems that can be solved by using coordinators (Section 2.1) and mediators (Section 2.2).

2.1 The Need for Coordinators: The Shared Resource Scenario

To better illustrate protocol coordination and the related underlying problems, in the following we describe the Shared Resource scenario. This explanatory example is concerned with the automatic assembly of a client-server component-based system. This system is formed by three components: two clients, respectively denoted as `C1` and `C2`, and one server denoted as `C3` (the component controlling the Shared Resource). This example, although very simple, exhibits coordination problems that exemplify the kind of problems that coordinator synthesis can solve. For instance, here, the problem is due to the presence of *race conditions* in accessing a shared resource.

Let us assume that we want to assemble a system formed by `C1`, `C2`, and `C3`. In doing so, we want to automatically prevent possible deadlocks and guarantee a specified coordination policy, hence, guaranteeing that the system's goal is reached.

Figure 1 represents the behavior of each component in terms of an LTS.

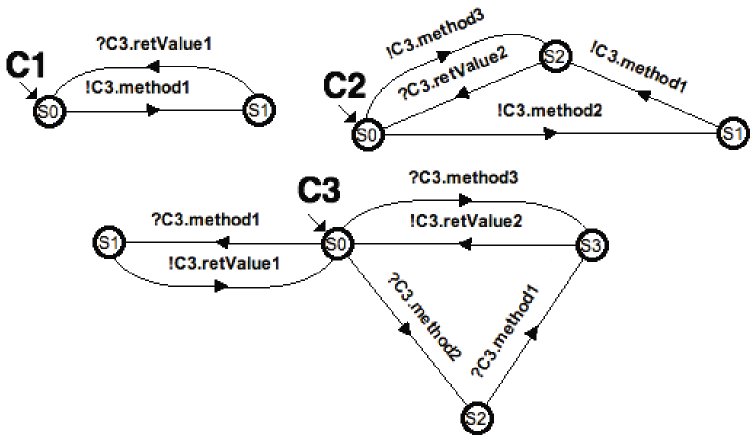


Fig. 1. Components' behavior for the Shared Resource scenario

Each LTS models the component observable behavior in an intuitive way. Each state of an LTS represents a state of the component and the state `S0` represents its initial state. Each action or complementary action performed by interacting with the environment of the component (i.e., all other components in parallel) is represented as a label of a transition into a new state. Actions are input or output. Within an LTS of a component, the label of an input action is prefixed by the question mark "?" (e.g., `?C3.retValue1` of `C1`). The label of an output action is prefixed by the exclamation mark "!" (e.g., `!C3.method2` of `C2`).

The interface of server `C3` exports three methods denoted as `C3.method1`, `C3.method2`, and `C3.method3`, respectively. While `C3.method2` has no return value, `C3.method1` and `C3.method3` can return some value. `C3.method1` returns two possible return values denoted as `C3.retValue1`, and `C3.retValue2`. The former is returned when a call of `C3.method1` has not preceded by a call of `C3.method2`. Otherwise, the latter is returned. `C3.method3` returns only one value, i.e., `C3.retValue2`. The two clients perform method calls according to the server interface.

It is worthwhile noticing that the described component interfaces syntactically match since either they already match or suitable component wrappers have been previously developed by the system assembler. As stated above, the problem of enabling communication is here considered as already solved. We recall that, in coordinator synthesis, the focus is on automatically preventing interaction protocol mismatches rather than enabling communication.

By continuing the description of our example, deadlocks can occur because of a race condition among `C1` and `C2`. In fact, one client (i.e., `C2`) performs a call of `C3.method2`, hence leading the server `C3` in a state in which it expects a call of `C3.method1`. While `C2` is attempting to perform the call of `C3.method1`, the other client (i.e., `C1`) performs such a call. In this scenario `C1`, `C2`, and `C3` are in the state `S1`, `S1`, and `S3` of their LTSs. Now, `C3` expects to return `C3.retValue2` as return value of `C3.method1` but `C2` is still waiting to perform a call of `C3.method1` and `C1` expects a different return value. Thus, a coordination mismatch occurs and it results in an deadlock in the interaction between `C1`, `C2`, and `C3`.

This mismatch can be solved by synthesizing a software coordinator that supervises the components' interaction by preventing the deadlock [73,8,72]. At the level of the coordinator's actual code, the coordinator is synthesized as a multi-threaded component that creates a thread for each request and for each caller performing such a request. Preventing, or solving if possible, deadlocks corresponds to put in a *waiting* state the thread that handles the request leading to the deadlock state and performed by the identified caller. Thus the coordinator will return, again, the control to the caller, for that request, only when it reaches a state in which the blocked request is allowable[1]. Such multi-threaded servers are supported by existing component technologies such as COM/DCOM or CORBA.

Another coordination issue that one can note is that, e.g., `C1` can always obtain the access to the shared resource, while `C2` never obtains it since `C2` can always require the access whenever the resource is already "lock" by `C1`. In other words, `C3` cannot be fair in providing the access to the shared resource it supervises. To solve this issue, a software coordinator can be automatically synthesized so as to enforce an *alternating protocol* policy [73] on the components' interaction. The coordinator allows only the alternating access of `C1` and `C2` to the shared resource.

2.2 The Need for Mediators: The Photo Sharing Scenario

To better illustrate protocol mediation and the related underlying problems, in the following we describe the Photo Sharing scenario within a stadium. In general, different versions of the Photo Sharing application may be available on the spectators' handhelds, thus calling for appropriate interoperability solutions.

Let us consider two Photo Sharing implementations: an Infrastructure-based (IB) and an ad hoc peer-to-peer (P2P) respectively shown by Figures 2 and 3. The protocols are depicted using LTSs where the name of actions are self-explanatory. We further use the convention that actions with overbar denote output actions while the ones with no overbar denote input actions.

[1] Meaning that, this time, that request performed from that caller does not lead to a deadlock.

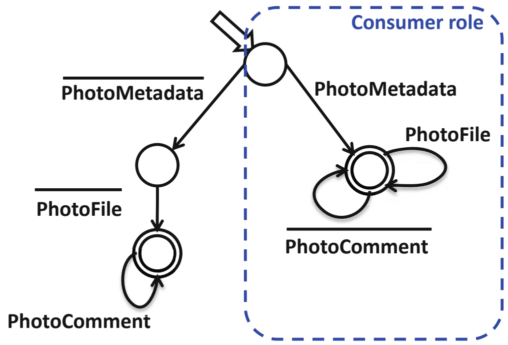


Fig. 2. Peer-to-Peer-based implementation

In the IB implementation, a Photo Sharing service is provided by the stadium, where only authenticated photographers are able to produce pictures while any spectator may download and even annotate pictures.

The P2P implementation allows for photo download, upload and annotation by any spectator, who are then able to directly share pictures using their handhelds.

Then, taking the producer perspective, the high level functionalities that the networked systems implement are: (1) the *authentication* -for the IB producer only- possibly followed by (2) the *upload of photo*, by sending both metadata and file, possibly followed by (3) the *download of comments*; on the other hand, taking the consumer perspective, the implemented high level functionalities are: (i) the *download of photo* by receiving both metadata and file respectively, possibly followed by (ii) the *upload of comments*.

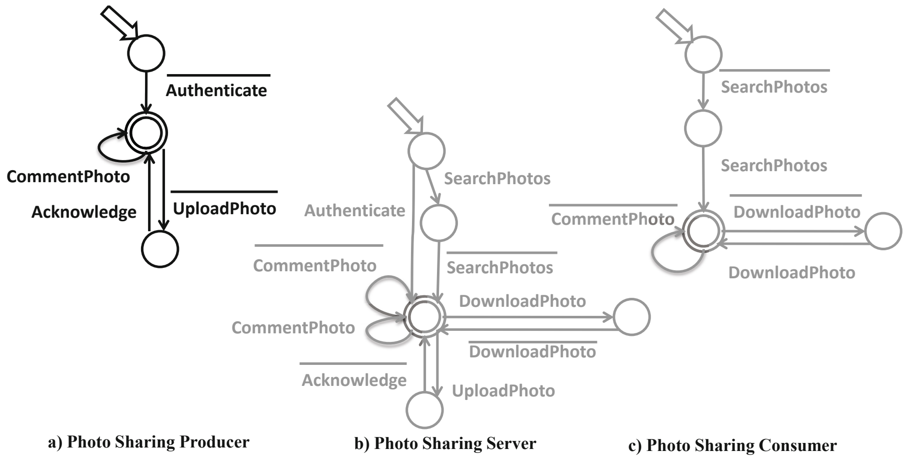


Fig. 3. Infrastructure-based implementation

In the P2P implementation, the networked system implements both roles of *producer* and *consumer*. Instead, while having similar roles and high level functionalities, the IB implementation differs with respect to the P2P one because: (i) in IB, the *consumer* and *producer* roles are played by two different/separate networked systems, in collaboration with the server, and (ii) comparing complementary roles among any P2P and IB, they have different interfaces and behaviors.

For the sake of illustration we consider as example the pair of mismatching applications made by: the IB producer (Figure 3 a)) and the P2P Photo Sharing consumer (portion within the dashed line of Figure 2). As can be noticed, the two protocols have different signatures and several discrepancies in the behavior that prevent their direct interoperability. Thus a mediator is needed to solve these heterogeneity in order to enable their communication. A detailed description of the mediator, including the problems it solves, is provided in Section 4.6 where the Photo Sharing scenario is used as running example.

3 Automatic Synthesis of Application-Layer and Failure-Free Coordinators

This section provides an overview of different approaches to the automatic synthesis of application-layer coordinators. We first introduce each approach by outlining their commonalities and differences. Then, through Sections 3.1 to 3.4, we give a complete overview of each approach.

Section 3.1 describes a method for the correct (with respect to coordination mismatches) and automatic assembly of component-based systems via centralized coordinator synthesis [73]. In this context, by considering communication issues already solved, the interoperability problem introduced in Section 1 can be rephrased as follows: *given a set of interacting components, C, and a set of behavioral properties, P, automatically derive a deadlock-free assembly, A, of these components which guarantees every property in P, if possible.* The assembly A is a composition of the components in C plus a synthesized coordinator. The coordinator is synthesized as an additional component which intercepts all the component interactions so as to control the exchange of messages with the aim of preventing possible deadlocks and those interactions that violate the properties in P. In [73] this problem is addressed by showing how to automatically synthesize the implementation of a centralized coordinator.

Unfortunately, in a distributed environment it is not always possible or convenient to introduce a centralized coordinator. For example, existing distributed systems might not allow the introduction of an additional component (i.e., the coordinator) which coordinates the information flow in a centralized way. Moreover, the coordination of several components might cause loss of information and bottlenecks hence slowing down the response time of the centralized coordinator. Conversely, building a distributed coordinator might extend the applicability of the approach to large-scale contexts.

To overcome the above limitations, in [8], an extension of the previous method is proposed. This extension is discussed in Section 3.2. The aim of the proposed extension is to automatically synthesize a distributed coordinator into a set of wrappers (local coordinators), one for each component whose interaction has to be controlled. The distributed coordinator synthesis approach has various advantages with respect to the synthesis of centralized coordinators. The most relevant ones are: (i) no centralized point of information flow exists; (ii) the degree of parallelism of the system without the coordinator is maintained; and (iii) all the domain-specific deployment constraints imposed on the centralized coordinator can be removed.

However, both methods are *static*; that is, if the system assembled by means of the synthesized coordinator evolves, e.g., a new component is added, or an existing one is either replaced or removed, the two methods have to be entirely re-performed in order to produce a new coordinator. Since, in the worst case, the computational complexity of the coordinator synthesis is exponential, re-performing the methods whenever a change in the systems occurs cannot be acceptable.

For this reason, in [57], a Software Architecture (SA) based method is proposed in which the usage of the system SA and of SA verification techniques allows the system assembler to design architectural components whose interaction is verified with respect to the specified properties. By exploiting this validation, the system assembler can perform coordinator synthesis by only focusing on each single architectural[2] component, hence refining it as an assembly of actual components which respect the architectural component observable behavior. In this way coordinator synthesis is performed locally on each architectural component, instead of globally on the whole system interactions, hence reducing the state-space explosion phenomenon due to the exponential complexity of the synthesis. The approach can be equally well applied to efficiently manage the whole reconfiguration of the system when one or more components need to be updated, still maintaining the required properties. The specified and verified system SA is used as starting point for the derivation of coordinators that are required to apply changes in the composed system. An overview of this approach is given in Section 3.3.

The methods outlined so far have been all applied to real case studies in the domains of COM/DCOM and J2EE applications. This experimentation has been carried on through the SYNTHESIS tool [6] that implements all the outlined methods.

All the previously mentioned methods do not account for the handling of non-functional attributes. Thus, recently, in [72], an extension of SYNTHESIS is proposed for automatically assembling real-time systems. The extended method and related tool, called SYNTHESISRT, are discussed in Section 3.4. This extension accounts for the handling of Quality-of-Service (QoS) attributes such as duration and latency of actions plus component clocks.

[2] It is an ideal component specified in the system SA.

3.1 Automatic Synthesis of Centalized Application-Layer and Failure-Free Coordinators

SYNTHESIS is a technique equipped with a tool [6] that permits to assemble a component-based application in a deadlock-free way [73,8]. Starting from a set of components Off The Shelf (OTS), SYNTHESIS assembles them together according to a so called coordinator-based architecture by synthesizing a coordinator that guarantees deadlock-free interactions among components. The code that implements the coordinator is automatically derived directly from the OTS (black-box) components' interfaces. Synthesis assumes a partial knowledge of the components' interaction behavior described as finite state automata plus the knowledge of a specification of the system to be assembled given in terms of Message Sequence Charts (MSCs) [4,74,75]. Furthermore, by exploiting that MSC specification, it is possible to go beyond deadlock. Actually, the MSC specification is an implicit failure specification. That is we assume to specify all the *desired* assembled system behaviors which are failure-free from the point of view of the system assembler, rather than to explicitly specify the failure. Under these hypotheses, SYNTHESIS automatically derives the assembling code of the coordinator for a set of components. The coordinator is derived in such a way to obtain a failure-free system. It is shown that the coordinator-based system is equivalent according to a suitable equivalence relation to the initial one once depurate of all the failure behaviors. The initial coordinator is a *no-op* coordinator that serves to model all the possible component interactions (i.e., the failure-free and the failing ones). Acting on the initial coordinator is enough to automatically prevent both deadlocks and other kinds of failure hence obtaining the failure-free coordinator.

As illustrated in Figure 4, the SYNTHESIS framework realizes a form of system adaptation. The initial software system is changed by inserting a new component, the coordinator, in order to prevent interactions failures.

The framework makes use of the following models and formalisms. An architectural model, the coordinator-based architecture that constrains the way components can interact, by forcing interaction to go through the coordinator. A set of behavioral models for the components that describe each single component's interaction behavior with the *ideal*[3] external context in the form of LTSs. A behavioral equivalence on LTS to establish the equivalence among the original system and the adapted/coordinated one. MSCs are used to specify the behavioral integration failure to be avoided, and then LTSs and *LTS synchronous product* [5,42] plus a notion of *behavioral refinement* [49] to synthesize the failure-free coordinator specification, as it is described in detail in [73]. As already mentioned, from the coordinator specification the actual code can then be automatically derived as either a centralized component [73] or a distributed one [8]. The latter is implemented as a set of *wrappers*, one for each component, that cooperatively realize the same behavior as the centralized coordinator. The next section gives an overview of this distributed coordinator synthesis approach.

[3] The one expected by the component's developer.

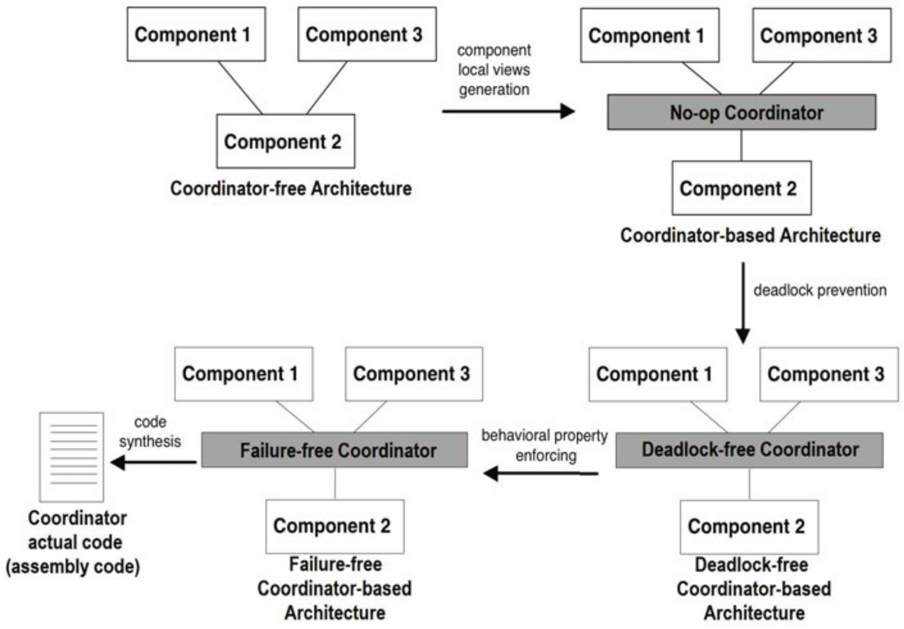

Component 1
Component 3
Component 2
Coordinator-free Architecture
component local views generation
Component 1
Component 3
No-op Coordinator
Component 2
Coordinator-based Architecture
deadlock prevention
Component 1
Component 3
Deadlock-free Coordinator
Component 2
Deadlock-free Coordinator-based Architecture
behavioral property enforcing
Component 1
Component 3
Failure-free Coordinator
Component 2
Failure-free Coordinator-based Architecture
code synthesis
Coordinator actual code (assembly code)

Fig. 4. Automatic synthesis of centralized failure-free coordinators

3.2 Automatic Synthesis of Distributed Application-Layer and Failure-Free Coordinators

As an extension of the method described in Section 3.1, the method that we discuss in this section assumes as input (see Figure 5): (*i*) a behavioral specification of the coordinator-free system formed by interacting components. It is given as a set $\{C_1, \ldots, C_n\}$ of LTSs (one for each component). The behavior of the system is modeled by composing in parallel all the LTSs and by forcing synchronization on common events; (*ii*) the specification of the desired behavior that the system must exhibit. This is given in terms of an LTS, from now on denoted by P_{LTS}.

These two inputs are then processed in two main steps:

1. By taking into account all component LTSs, we automatically derive the LTS that models the behavior of a centralized deadlock-free coordinator. This first step is inherited from the approach described in Section 3.1 for the synthesis of centralized coordinators. Whenever P_{LTS} ensures itself deadlock-freeness and its traces are all traces of the centralized coordinator LTS, such a step is not required and, hence, the centralized coordinator cannot be generated. We recall that, at the worst case, the synthesis of the centralized coordinator has an exponential computational complexity in the maximum number of states of the component LTSs. By avoiding the generation of the centralized coordinator, the method's complexity becomes polynomial in the number of states of P_{LTS}. The first step terminates

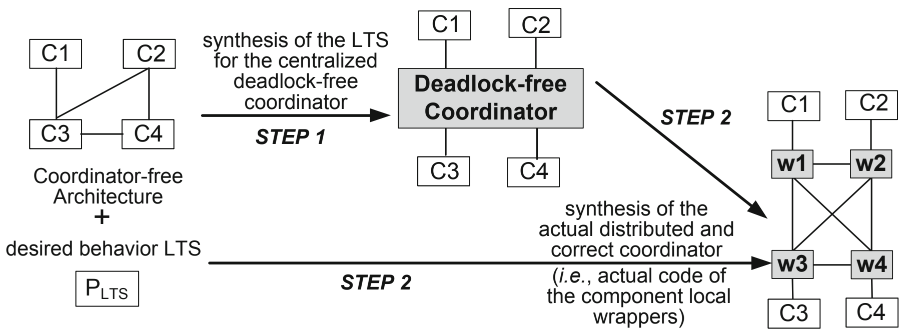

C1
C2
C3
C4
Coordinator-free Architecture
+
desired behavior LTS
P_LTS
synthesis of the LTS for the centralized deadlock-free coordinator
STEP 1
Deadlock-free Coordinator
STEP 2
synthesis of the actual distributed and correct coordinator
(i.e., actual code of the component local wrappers)
w1
w2
w3
w4

Fig. 5. Automatic synthesis of distributed failure-free coordinators

by checking whether enforcing P_{LTS} is possible or not. This check is implemented by a suitable notion of refinement. Refinement, in general, formalizes the relation between two LTSs at different level of abstractions. Refinement is usually defined as a variant of *simulation*. In our method, we use a suitable notion of *strong simulation* [49] to check a refinement relation between two LTSs.

2. In the second step, let K be the LTS of the centralized coordinator. If K has been generated and it has been checked that P_{LTS} can be enforced on it, our method explores K looking for those states representing the *last chance* before entering an execution trace that leads to a deadlock. For instance, in Figure 6, the state `S4` represents the last chance state before incurring in the deadlock state `S7`. This information is crucial for deadlock prevention purposes.

   The search of the last chance states is realized by means of a depth-first search, performed on K, whose aim is to save those states into the local wrappers of the components that could lead the system from a last chance state to a deadlock by means of a so called *critical action*. The idea is therefore not to allow a component to perform a critical action before being sure that the system will not reach a deadlock state. By interacting with the Synthesis tool, the user can tag component actions as either *controllable* or *uncontrollable* by the external environment. If such a critical action is controllable then it can be discarded. Otherwise, if it is uncontrollable, Synthesis performs a *controller synthesis step* [60,16] that "backtracks" by looking for the first controllable action that can be discarded to prevent the execution of the critical action. After the execution of this depth-first search on K, the set of last chance states and associated critical actions are stored in a table, one for each component wrapper.

   The second step also explores P_{LTS} to retrieve information crucial for undesired behavior prevention. The aim here is to split and distribute P_{LTS}

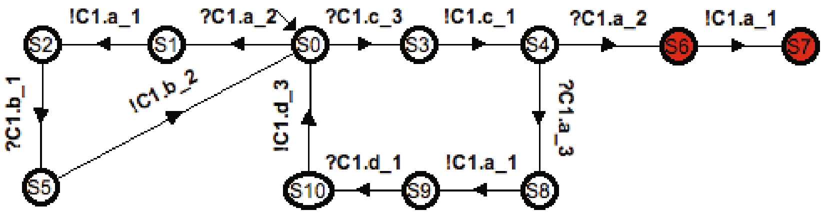


Fig. 6. An example of a centralized coordinator LTS in SYNTHESIS

in a way that each local wrapper knows which actions the wrapped component is *allowed* to execute. This is realized by means of a depth-first search on P_{LTS}.

Referring to Figures 6 and 7 for instance, the wrapper of component $C3$ must not allow the component to send the request `C1.a`, if the current global state of the system matches the state `S0` in P_{LTS}, hence enforcing the desired behavior modeled by P_{LTS}. In particular, the label $\{$`!−C1.a_2,!−C1.a_3`$\}$ of the loop on `S0` denotes two loops, one labeled with `!−C1.a_2` and one labeled with `!−C1.a_3`. The action `!C1.a_3` denotes an output action `C1.a` by `C3`; `!−C1.a_3` represents *its neagation*, i.e., all possible actions different from it.

The sets of *last chance states* and *allowed actions* are stored and, subsequently, used by the local wrappers as basis for correctly synchronizing with each other by exchanging additional communication. In other words, the local wrappers interact with each other to restrict the components' standard communication (modeled by K) by allowing only the part of the communication that is correct with respect to deadlock-freeness and P_{LTS}. By decentralizing K, the local wrappers preserve parallelism of the components forming the system.

The message exchange among wrappers for synchronization purposes is realized by means of the two procedures *Ask* and *Ack*, whose implementation is automatically synthesized by SYNTHESIS. The first is used to ask the permission to the other wrappers before allowing a component to proceed with a critical action. The second is used to reply to a message sent by procedure *Ask* when the global state is safe.

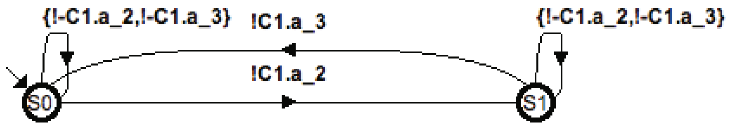


Fig. 7. An example of a desired behavior LTS in SYNTHESIS

3.3 Automatic Synthesis of Application-Layer Coordinators for Evolvable Systems

This coordinator synthesis method is composed of four main phases organized as shown in Figure 8. In the following, the description of the method assumes that an SA has been modeled by using the Charmy framework [56] (*System SA + properties of interest* in Figure 8).

Design-time phase. the first phase concerns the system SA verification. This phase is performed by using Charmy. The input of this phase is an SA and the properties that one wants to check. The output is a system SA specification that respects the properties of interest (*Verified system SA* in Figure 8).

For each verified architectural component that has not yet been implemented, the *Actual components selection phase* is performed. After that all the architectural components have been implemented, they are deployed (*Re-implemented components deployment* in Figure 8) hence producing a first running version of the system (*Running system* in Figure 8).

Actual components selection phase. our method implements each architectural component as an assembly of actual components acquired from a third-party, when possible. This phase aims at selecting third-party components

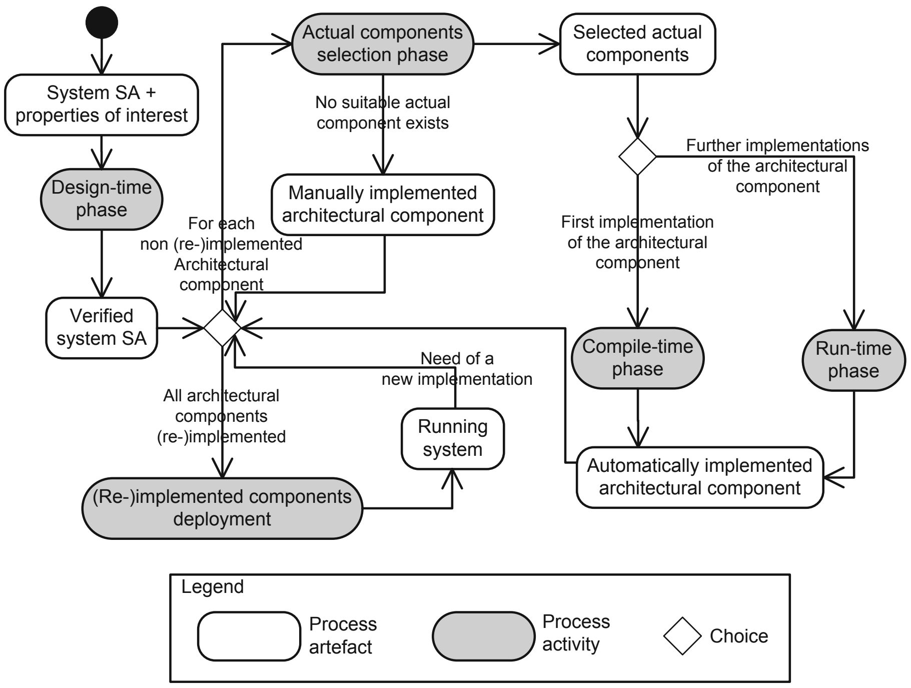

System SA + properties of interest
Design-time phase
Verified system SA
For each non (re-)implemented Architectural component
Actual components selection phase
No suitable actual component exists
Manually implemented architectural component
Selected actual components
Further implementations of the architectural component
First implementation of the architectural component
Compile-time phase
Run-time phase
Automatically implemented architectural component
Need of a new implementation
Running system
All architectural components (re-)implemented
(Re-)implemented components deployment
Legend
Process artefact
Process activity
Choice

Fig. 8. Automatic synthesis of failure-free coordinators for evolvable systems

by looking at their interfaces and functionalities. For the selection criteria used to establish which actual components have to be acquired to implement an architectural component, we refer to [57] where the method is discussed in detail. This phase takes as input a verified architectural component and it is performed with respect to a repository of actual components acquired from a third-party [54] (black-box components). The output is the set of actual components selected as possible candidates for the implementation of the architectural one or an empty set. In case of an empty set, the architectural component is manually implemented (*Manually implemented architectural component* in Figure 8) since we did not find suitable components that can be assembled to implement the considered architectural component. In this case it is up to the developer to guarantee that the component implementation conforms to its architectural specification, e.g., via verification techniques.

If possible candidates are found (*Selected actual components* in Figure 8) they could still need some adaptations (e.g., they might provide more functionalities as needed or interaction mismatches might occur). The compile-time phase and the run-time phase will automatically manage that in the first implementation of the architectural component and in its further implementations, respectively.

Compile-time phase. in order to correctly implement the considered architectural component, this phase automatically produces an assembly of the selected actual components that is correct with respect to the architectural component's observable behavior (*Automatically implemented architectural component* in Figure 8). This is done by exploiting the SYNTHESIS tool as either described in Sections 3.1 or 3.2 depending on which kind of implementation is required for the architectural component, centralized or distributed.

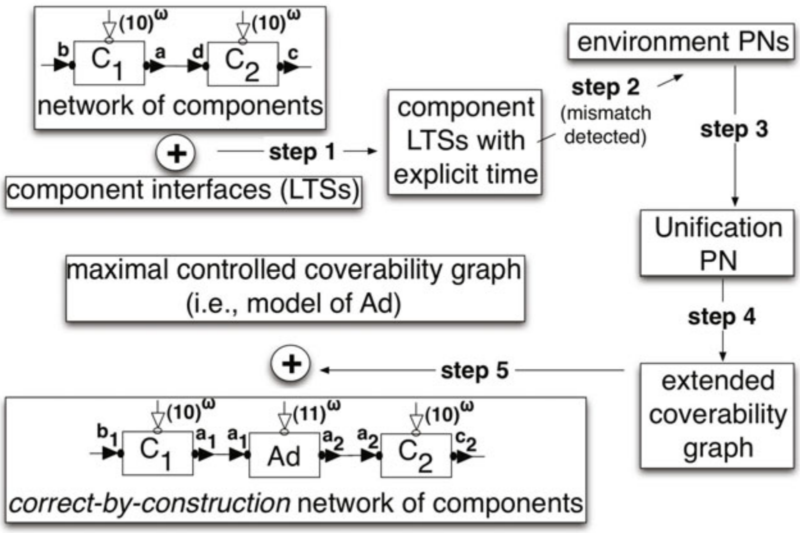
Run-time phase. when a new implementation of an architectural component is needed (the transition *Need of a new implementation* outgoing from *Running system*), the correct (re-)implementation of the considered architectural component is produced analogously to what is done in the compile-time phase, i.e., again via the SYNTHESIS tool. The run-time phase performs additional operations with respect to the compile-time phase. These operations are the suspension of the running system in a consistent state and the transfer of the computational state.

3.4 Automatic Synthesis of Application-Layer Coordinators for Real-Time Systems

Recently, the SYNTHESIS approach and its related tool has been extended to the context of real-time systems [72]. This extension, hereafter called SYNTHESISRT, has been developed by the Software Engineering research group at University of L'Aquila in cooperation with the POP ART project team at INRIA Rhône-Alpes. In [72], it is shown how to deal with the compatibility, communication, and QoS issues that can raise while building a real-time system from reusable black-box components within a lightweight component model where components follow a data-flow interaction model. Each component declares input and output ports which are the points of interaction with other components and/or the execution environment. Input (resp., output) ports of a component are connected to output (resp., input) ports of a different component through synchronous links. Analogously to the version of SYNTHESIS without real-time constraints, a component interface includes a formal description of the *interaction protocol* of the component with its expected environment in terms of sequences of writing and reading actions to and from ports. The interface language is expressive enough to specify QoS constraints such as writing and reading *latency*, *duration*, and *controllability*, as well as the component's *clock* (i.e., its activation frequency). In order to deal with incompatible components (e.g., clock inconsistency, read-/write latency/duration inconsistency, mismatching interaction protocols, etc.) we synthesize coordinators interposed between two or more interacting components. A coordinator is a component that mediates the interaction between the components it supervises, in order to harmonize their communication. Each coordinator is automatically derived by taking into account the interface specification of the components it supervises. The coordinator synthesis allows the developer to automatically and *incrementally* build *correct-by-construction* systems from third-party components.

Figure 9 shows the main steps of the method performed by SYNTHESISRT by also highlighting the used formalisms/models.

We take as input the architectural specification of the network of components to be composed and the component interface specifications. The behavioral models of the components are generated in form of LTSs that make the elapsing of time explicit (step 1). Connected ports with different names are renamed such that complementary actions have the same label in the component LTSs (see



(10)^ω
(10)^ω
b
C1
a
d
C2
c
network of components
+
step 1
component interfaces (LTSs)
component LTSs with explicit time
step 2 (mismatch detected)
environment PNs
step 3
Unification PN
step 4
extended coverability graph
step 5
maximal controlled coverability graph (i.e., model of Ad)
+
(10)^ω
(11)^ω
(10)^ω
b1
C1
a1
a1
Ad
a2
a2
C2
c2
correct-by-construction network of components

Fig. 9. Main steps of the coordinator synthesis for real-time components

actions a and d in Figure 9). Possible mismatches/deadlocks are checked by looking for possible sink states into the parallel composition of the LTSs. The coordinator synthesis process starts only if such deadlocks are detected.

The synthesis first proceeds by constructing a *Petri net* (PN) [52] representation of the environment expected from a component in order not to block it (step 2). It consists in complementing the actions in the component LTSs that are performed on connected ports, considering the actions performed on unconnected ports as internal actions. Moreover, a buffer storing read and written values is modeled as a place in the environment PN for each IO action. Each such PN represents a partial view of the coordinator to be built. It is partial since it reflects the expectation of a single component. In particular, a write (resp. read) action gives rise to a place (buffer) without outgoing (resp. incoming) arcs.

The partial views of the coordinator are composed together by building causal dependencies between the reading/writing actions and by unifying time-elapsing transitions (step 3). Furthermore, the places representing the same buffer are merged in one single place. This *Unification PN* models a coordinator that solves deadlocks using buffers to desynchronize received events from their emission.

However, the unification PN may be not completely correct, in the sense that it can represent a coordinator that may deadlock and/or that may require unbounded buffers. In order to obtain the most permissive and correct coordinator, we generate an extended version of the graph usually known in PNs theory [52] as the coverability graph [29] (step 4).

Our method automatically restricts the behavior of the coordinator modeled by the extended coverability graph in order to keep only the interactions that are deadlock-free and that use finite buffers (i.e., bounded interactions). This is done by automatically constructing, if possible, an "instrumented" version of our extended coverability graph, called the *Controlled Coverability Graph (CCG)*. The CCG is obtained by pruning from the extended coverability graph both the *sinking* paths and the *unbounded* paths, by using a *controller synthesis* step [61] (step 5). `Ad`, in the figure, denotes the synthesized coordinator.

This process also performs a *backwards error propagation* step in order to correctly take into account the case of sinking and unbounded paths originating from the firing of uncontrollable transitions.

If it exists, the maximal CCG generated is the LTS modeling the behavior of the correct (i.e., deadlock-free and bounded) coordinator. This coordinator models the correct-by-construction assembly code for the components in the specified network. If it does not exist, a correct coordinator assembling the components given as input to our method cannot be automatically derived, and hence our method does not provide any assembly/coordination code for those components.

4 Automatic Synthesis of Application-Layer Mediators

This section describes our recent work on the automatic synthesis of application-layer mediators. We overview our methodology in Section 4.1 and we give formal foundations in Section 4.2. Then we provide the formalization of our theory by respectively presenting the protocol abstraction in Section 4.3, protocol matching in Section 4.4, and protocol mapping in Section 4.5. Finally we illustrate the application of the theory to the Photo Sharing scenario in Section 4.6. Abstraction, matching, and mapping are fundamentals operations of our mediator synthesis approach.

As already illustrated in Section 1, we focus on the interoperability problem between heterogeneous protocols within the UbiComp environment. For the sake of simplicity, and without loss of generality, we limit the number of protocols to two but the work can be generalized to an arbitrary number of protocols.

In particular, we focus on **compatible** or **functionally matching protocols**. Functional matching means that heterogeneous protocols can *potentially communicate* by performing *complementary sequences of actions* (or *complementary conversations*).

Potentially means that communication may not be achieved because of *mismatches* (heterogeneity), i.e., the languages of the two protocols are different, although semantically equivalent. For example, protocol languages can have: (i) different granularity, or (ii) different alphabets. Protocols behavior may have, for example, different sequences of actions because of (a.1) the order in which actions are performed by a protocol is different from the order in which the other protocol performs the same actions; (a.2) interleaved actions related to *third parties communications* i.e., other systems, the environment. In some cases, as for example (i), (ii) and (a.1), it is necessary to properly perform a manipulation of the two languages. In the case (a.2) it is necessary to provide an abstraction of the two actions sequences that results in sequences containing only actions that are relevant to the communication.

Communication is then possible if the two possibly manipulated (e.g., reordered) and abstracted sequences of actions are complementary, i.e., are the same sequences of actions while having opposite output/input "type" for all actions.

Therefore, the problem we address and overcome, is the *interoperability between heterogeneous protocols in the UbiComp environment.*

With **interoperability**, we mean the property referring to the ability of heterogeneous protocols *to communicate and coordinate* to reach their goal(s). Communication and coordination are expressed by synchronization, i.e., two systems succeed in coordinating if they are able to synchronize hence reaching their goal(s).

In order to make communication between heterogeneous protocols possible, we proposed as **solution** *a theory of mediators* [36,67,64] that we revise and extend in the remainder of this section. The theory, reasoning about the mismatches of the compatible protocols, automatically elicits and synthesizes an *emerging mediator* that solves them allowing protocol interoperability. The theory paves the way for run-time (or on-the-fly) approaches to the mediators synthesis.

A **mediator** is then a protocol that allows the communication among compatible protocols by mediating their differences.

We *assume* that each device, e.g. PDA, smartphone, or tablet, is equipped, for its applications, with the (i) behavioral specification and their (ii) semantical characterization of their actions through ontologies. Taking the perspective of two systems that have compatible protocols and that also communicate with third parties, we assume that there exists also (iii) the proper environment for them, i.e., the other systems representing third parties. Further, we concentrate on application layer interoperability while assuming solved the heterogeneity of the underlying layers.

4.1 Towards Emerging Mediators

Figure 10 depicts the main elements of our methodology which we describe in the following.

The method includes:

(i) Two application-layer protocols P and Q whose representation is given in terms of LTSs, where the *initial* and *final states* on the LTSs define the *sequences of actions* (traces) that characterize the *coordination policies* of the protocols.
(ii) Two *ontologies* O_P and O_Q describing the meaning of P and Q's actions, respectively.
(iii) Two *ontology mapping functions* $maps_P$ and $maps_Q$ defined from O_P and from O_Q to a common ontology. The intersection O_{PQ} on the common ontology identifies the "common language" between P and Q. For simplicity, and without loss of generality, we consider protocols P and Q that have disjoint languages and that are minimal where we recall that every finite LTS has a unique minimal representative LTS.

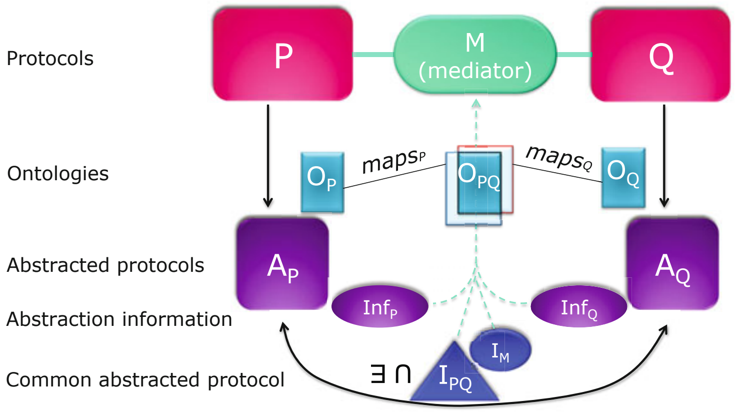


Fig. 10. An overview of our approach

(iv) Then, starting from P and Q, and based on the ontology mapping, we build two abstractions A_P and A_Q by relabeling P and Q, respectively, where the actions not belonging to the common language O_{PQ} are hidden by means of silent actions (τs); moreover, we store some abstraction information (i.e., used to make the abstraction), Inf_P and Inf_Q, that in case of positive matching check, will be exploited to synthesize the mediator during the mapping;

(v) Then, we check the compatibility of the protocols by looking for complementary traces (the set I_{PQ} in figure), modulo mismatches and third parties communications, between the sets of traces T_P and T_Q generated by A_P and A_Q, respectively. If this is the case, then we are able to synthesize a mediator that makes it possible for the protocols to coordinate. Hence, we store the matching information (i.e., used to make the abstraction) I_M that will be exploited during the mapping.

(vi) Finally, given two protocols P and Q, and an environment E, the mediator M that we synthesize is such that when building the parallel composition $P||Q||E||M$, P and Q are able to coordinate by reaching their final states under the hypothesis of fairness.

4.2 Formal Foundations

The application-layer interaction protocol, as described in Section 1, is the behavior of a system in terms of the actions it exchanges with other application-layer interaction protocols. In this section, a characterization of such protocols is provided together with a conceptualization of the application actions.

Protocols as LTS

As mentioned in Section 2, we use LTSs to characterize the protocols. LTSs constitute a widely used model for concurrent computation and are often used as a semantic model for formal behavioral languages such as process algebras. Let *Act* be the set of observable actions (input/output actions), we get the following definition for LTS:

Definition 1 (*LTS*). *A LTS P is a quadruple* (S, L, D, s_0) *where:*
S *is a finite set of states;*
$L \subseteq Act \bigcup \{\tau\}$ *is a finite set of labels (that denote observable actions) called the alphabet of P.* τ *is the silent action. Labels with an overbar in L denote output actions while the ones without overbar denote input actions. We also use the convention that for all* $l \in L, \overline{\overline{l}} = l$[4].
$D \subseteq S \times L \times S$ *is a transition relation;*
$s_0 \in S$ *is the initial state.*

We then denote with $\{L \bigcup \{\tau\}\}^*$ the set containing all words on the alphabet L. We also make use of the usual following notation to denote transitions:
$s_i \xrightarrow{l} s_j \Leftrightarrow (s_i, l, s_j) \in D$

[4] We inherit this convention from *Calculus of Communicating Systems* (CCS) [49].

We consider an extended version of LTS, where the set of the LTS' *final states* is explicit. An **extended LTS** is then a quintuple (S, L, D, F, s_0) where the quadruple (S, L, D, s_0) is a LTS and $F \subseteq S$. From now on, we use the terms LTS and extended LTS interchangeably, to denote the latter one.

The initial state together with the final states, define the boundaries of the protocol's coordination policies. A **coordination policy** is indeed defined as any trace that starts from the initial state and ends into a final state. It captures the most elementary behaviors of the NS which are meaningful from the user perspective (e.g., upload of photo of photo sharing producer meaning upload of photo followed by the reception of one or more comments). Then, a coordination policy represents a communication (i.e., coordination or synchronization) unit. We get the following formal definition of traces/coordination policy:

Definition 2 (*Trace* or *Coordination Policy*). *Let* $P = (S, L, D, F, s_0)$. *A trace* $t = l_1, l_2, \ldots, l_n \in L^*$ *is such that:*
$\exists (s_0 \xrightarrow{l_1} s_1 \xrightarrow{l_2} s_2 \ \ldots \ s_m \xrightarrow{l_n} s_n)$ *where* $\{s_1, s_2, \ldots, s_m, s_n\} \in S \wedge s_n \in F$.

We use the usual compact notation $s_0 \overset{t}{\Rightarrow} s_n$ to denote a trace, where t is the concatenation of actions of the trace.

Moreover we define a **subtrace** as any sequence in a protocol (it may be also a trace). More formally:

Definition 3 (*Subtrace*). *Let* $P = (S, L, D, F, s_0)$. *A subtrace* $st = l_i, l_{i+1}, \ldots, l_n \in L^*$ *is such that:*
$\exists (s_i \xrightarrow{l_i} s_{i+1} \xrightarrow{l_{i+1}} s_{i+2} \ \ldots \ s_m \xrightarrow{l_m} s_n)$ *where* $\{s_i, s_{i+1}, s_{i+2}, \ldots, s_m, s_n\} \in S$

Similarly to traces, also in this case we use the compact notation $s_i \overset{st}{\Rightarrow} s_n$.

LTSs can be combined using the LTS parallel composition operator. Several semantics have been given in the literature for this operator. The one needed here is similar to the one of CSP (*Communicating Sequential Processes*) [63]: protocols P and Q synchronize on complementary actions while proceeding independently when engaged in non complementary actions. Moreover, we need a *synchronous* reference model as the one of CSP or FSP (*Finite State Process*) [47] where the synchronization is forced when an interaction is possible. Differently, the asynchronous model like the one of CCS [49], would allow agents to non-deterministically choose to not interact by performing complementary actions a and $\overline{a}$ separately.

Although the semantics and the model we need are *à la* CSP, we use CCS because (i) it is able to emulate the synchronous model of CSP thanks to the restriction operator and (ii) it has several characteristics that CSP does not have and that we need, e.g., complementary actions and τs .

Then our parallel composition semantics is that protocols P and Q synchronize on complementary actions producing an internal action τ in the parallel composition. Instead, P and Q can proceed independently when engaged in non complementary actions. An action of P (Q resp.) for which no complementary action exists in Q (P resp.), is executed only by P (Q resp.), hence, producing the same action in the parallel composition.

Definition 4 (*Parallel composition of protocols*). *Let* $P = (S_P, L_P, D_P, F_P, s_{0_P})$ *and* $Q = (S_Q, L_Q, D_Q, F_Q, s_{0_Q})$. *The parallel composition between* P *and* Q *is defined as the LTS* $P||Q = (S_P \times S_Q, L_P \cup L_Q, D, F_P \cup F_Q, (s_{0_P}, s_{0_Q}))$ *where the transition relation* D *is defined as follows:*

$$\frac{P \xrightarrow{m} P'}{P|Q \xrightarrow{m} P'|Q} \qquad (where\ m \in L_P \wedge \overline{m} \notin L_Q)$$

$$\frac{Q \xrightarrow{m} Q'}{P|Q \xrightarrow{m} P|Q'} \qquad (where\ m \in L_Q \wedge \overline{m} \notin L_P)$$

$$\frac{P \xrightarrow{m} P'; Q \xrightarrow{\overline{m}} Q'}{P|Q \xrightarrow{\tau} P'|Q'} \qquad (where\ m \in L_P \wedge \overline{m} \in L_Q)$$

Note that when we build the parallel composition of protocols P, Q, with the environment E, and the mediator M, the composed protocol $P|Q|E|M$ is restricted to the language made by the union of the common languages between each pair of protocols. Thus, this restriction force all the protocols to synchronize when an interaction is possible among them.

Ontologies

Ontologies play an important role in realizing connectors which primarily relies on reasoning about systems functionalities. More in detail, what is needed is to identify matching sequences of observable actions among the actions performed by the systems. Ontologies play a key role in identifying such matching and allow overcoming the inherent heterogeneity of NSs.

In the literature, [40,39] the ontologies and the ontology mapping are defined as follows:

- "an *ontology* is a pair $O = (S, A)$, where S is the (ontological) signature describing the vocabulary and A is a set of (ontological) axioms specifying the intended interpretation of the vocabulary in some domain of discourse".
- "A *total ontology mapping* from $O_1 = (S_1, A_1)$ to $O_2 = (S_2, A_2)$ is a morphism
  $f : S_1 \rightarrow S_2$ of ontological signatures, such that, $A_2 = f(A_1)$, i.e., all interpretations that satisfy O_2's axioms also satisfy O1's translated axioms".

Towards enabling mediators, in the next section we will detail application ontologies characterizing the application actions.

4.3 Abstraction Formalization

Given the definition of extended LTS associated with two interaction protocols run by NSs, we want to identify whether such two protocols are *functionally* *matching* and, if so, to synthesize the mediator that enables them to interoperate, despite behavioral mismatches and third parties communications.

We recall that with *functional matching*, we mean that given two systems with respective interaction protocols P and Q, ontologies O_P and O_Q describing their actions, ontology mapping functions $maps_P$ on P and $maps_Q$ on Q, and their intersecting common ontology O_{PQ}, there exists *at least one pair of complementary traces* (with one trace in P and one in Q) that allows P and Q to coordinate. In other words, one or more sequences of actions of one protocol can synchronize with one or more sequences of actions in the other. This can happen by properly solving mismatches, using the basic patterns discussed in [66,65], and managing communications with third parties. Thus, we expect to find, at a given level of abstraction, a common protocol C that represents the potential interactions of P and Q. This leads us to formally analyze such alike protocols to find - if it exists - C and a suitable mediator that allows the interoperability that otherwise would not be possible. This problem can be formulated as a kind of anti-unification problem [35,58,79,62].

In order to find the protocols' abstractions, we exploit the information contained in the ontology mapping to suitably relabel the protocols. Specifically, as detailed in the following, the relabeling of LTSs produces new LTSs that are labeled only by common actions and τs, and hence are more abstract than before (e.g., sequences of actions may have been compressed into single actions). For illustration, Figure 11 summarizes the ontological information of the IB Producer of Figure 3 a) (first column) and of the P2P Photo Sharing of Figure2 (third column). The second column shows their *common language*. We recall that: (1) the overlined actions are output/send action while non-overlined are input/receive; (2) the P2P application implements both roles, producer and consumer, while the IB application we are focusing on, is the producer role only(the overall Photo Sharing is implemented by three separate IB applications). This explains why we have in the table two non-paired actions; because they are paired with the actions of the other IB applications.

Infrastructure-based Photo-Sharing Producer	**Common Language Projected on the Protocols**		**Peer-to-peer Photo-Sharing version 1**
$\overline{\text{UploadPhoto.}}$ Acknowledge	$\overline{\textit{UP}}$ *(upload photo)*	*UP* *(download photo)*	PhotoMetadata. PhotoFile
CommentPhoto	*UC* *(download comment)*	$\overline{\textit{UC}}$ *(upload comment)*	$\overline{\text{PhotoComment}}$
-	-	$\overline{\textit{UP}}$ *(upload photo)*	$\overline{\text{PhotoMetadata.}}$ $\overline{\text{PhotoFile}}$
-	-	*UC* *(download comment)*	PhotoComment

Fig. 11. Ontology mapping between Infrastructure-based Photo Sharing Producer and peer-to-peer Photo Sharing (Figure 3 a) and Figure2 respectively)

In the following we describe more formally the abstraction step. We specialize the definition of total ontology mapping of Section 4.2, that maps single elements of S_1 into single elements of S_2, by defining an *abstraction ontology mapping* that maps the S_1 language (i.e., S_1^*) into S_2, i.e., $maps : S_1^* \rightarrow S_2$.

We use such specialized ontology mapping on the ontologies of the compatible protocols, where the vocabulary of the source ontology is the language of the protocol. More formally:

Definition 5 (*Abstraction Ontology Mapping*). *Let:*

- $P = (S_P, L_P, D_P, F_P, s_{0_P})$,
- $O_P = (L_P^*, A_P)$ *be the ontology of* P,
- $O = (L, A)$ *be an ontology referred as abstract ontology,*
- $st \in L_P^*$ *be a subtrace on* P.

The abstraction ontology mapping is a function maps such that:
$maps : L_P^* \rightarrow L$.

The application of the above abstraction ontology mapping *maps* on the ontology of P returns as result the set L_{abs} of labels on the abstract ontology defined as $L_{abs} = \{l \in L : \forall\ st \in L_P^*\ \ l = maps(st)\}$.

The abstract protocols are then obtained, leveraging on the abstraction ontology mapping, by relabeling protocols with labels of their common language and τs for the thirds parties languages. A necessary condition for the existence of a common language between two protocols P and Q, is that there exist two abstraction ontology mapping $maps_P$ on P and $maps_Q$ on Q that map the languages L_P^* of P and L_Q^* of Q into the same/common abstract ontology. Thus, to identify the common language, we first map each protocol's ontology into a common ontology and then by intersection, we find their common language.

Operationally, we do not work on protocols while we reason on traces: starting from a protocol P (Q resp.), we extract all the traces from it and apply the relabelling on the traces that result into a set of abstracted traces, with labels belonging to the common language and τs. However, the abstract protocol(s) of P (Q resp.), can be easily obtained by merging the traces where possible (e.g. common prefixes). Similarly, we use a reasoning on traces also for the matching and mapping phases.

It has to be noticed that the set of all the traces may not be finite. Then, the abstraction ontology mapping can be applied to an infinite set of traces. We consider minimal protocols[5]. Hence, the infinite set of traces is represented by a minimal automaton (containing at least a final state). Then, the abstraction ontology mapping on such minimal automaton, either applies directly (to the minimal automaton) returning a set of (abstracted) traces on the common language and τs, or it does not exist any automaton unfolding on which the abstraction ontology mapping applies.

[5] This is similar to the normal form of a system of recursive equations in [37] which is based on the idea to eliminate repetitions of equivalent recursive equations (that is equations with the same unfolding).

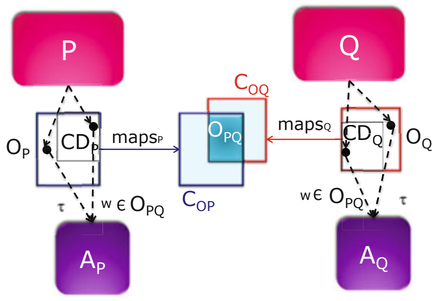


Fig. 12. The abstract protocol building

Figure 12 depicts the abstraction of the protocols. Let us consider two minimal and deterministic protocols P and Q with their respective ontologies $O_P = (L_P^*, A_P)$ and $O_Q = (L_Q^*, A_Q)$ and their abstraction ontology mappings $maps_P$ and $maps_Q$ respectively.

We first map O_P and O_Q, through $maps_P : L_P^* \rightarrow L$ and $maps_Q : L_Q^* \rightarrow L$ respectively, into a common ontology $O = (L, A)$ where C_{OP} and C_{OQ} represent the codomain sets of $maps_P$ and $maps_Q$ respectively.

The *common language* between P and Q is defined as the intersection O_{PQ} of C_{OP} and C_{OQ}. In particular, it is built by: (1) applying the abstraction ontology mapping to P and Q respectively thus obtaining the two sets of labels C_{OP} and C_{OQ} respectively; (2) starting from pairs of actions l and $\overline{l}$ ($\overline{l}, l$ resp.) belonging to C_{OP} and C_{OQ} respectively, storing into O_{PQ} the action l - without taking into account the type send/receive. Below, we define the common language more formally:

Definition 6 (*Common Language*). *Let:*

- $P = (S_P, L_P, D_P, F_P, s_{0_P})$ *and* $Q = (S_Q, L_Q, D_Q, F_Q, s_{0_Q})$,
- st_P, st_Q *be subtraces of* P *of* Q *respectively,*
- $O_P = (L_P^*, A_P)$ *be the ontology of* P *and* $O_Q = (L_Q^*, A_Q)$ *be the ontology of* Q,
- $O = (L, A)$ *be an ontology,*
- $maps_P : L_P^* \rightarrow L$ *be the abstraction ontology mapping of* P *and* $maps_Q : L_Q^* \rightarrow L$ *be the abstraction ontology mapping of* Q,

The common language O_{PQ} *between* P *and* Q *is defined as:*
$O_{PQ} = \{l : l \ (or\ \overline{l}) = maps_P(st_P) \wedge l \ (or\ \overline{l}) = maps_Q(st_Q)\ \}$
where st_P, st_Q *implement basic mismatches (as defined in papers [66,65]).*

For instance, the pairs of labels ($\overline{UP}$, UP), or (UP, UP), or (UP, $\overline{UP}$), or ($\overline{UP}$, $\overline{UP}$) let us derive UP as an action belonging to the common language.

The abstract protocol A_P (A_Q resp.) of P (Q resp.), is built as follows: for each trace t_P of P (t_Q of Q resp.) build a new trace t'_P (t'_Q) such that:

1. for each chunk (sequences of states and transitions) of t_P (t_Q resp.) labeled by subtraces on D_P (D_Q resp.), build a single transition in t'_P (t'_Q) labeled with a label on O_{PQ};
2. for all the other chunks of t_P (t_Q resp.) labeled with actions belonging to the thirds parties language, build chunks labelled with τs.

In the following we define more formally the relabeling function that we exploit:

Definition 7 (*Relabelling function*). *Let:*

- $P = (S_P, L_P, D_P, F_P, s_{0_P})$ *and* $Q = (S_Q, L_Q, D_Q, F_Q, s_{0_Q})$ *be protocols,*
- $O_P = (L^*_P, AX_P)$ *and* $O_Q = (L^*_Q, AX_Q)$ *be ontologies of* P *and* Q *respectively,*
- $O = (L, A)$ *be a common ontology for* P *and* Q*,*
- $maps_P : L^*_P \to L$ *and* $maps_Q : L^*_Q \to L$ *be abstraction ontology mappings of* P *and* Q *respectively,*
- C_{OP} *and* C_{OQ} *be the codomain sets of* $maps_P$ *and* $maps_Q$ *respectively,*
- O_{PQ} *be the common language between* P *and* Q*.*

The relabeling function relabels is defined as: relabels : $(P, maps_P, O_{PQ}) \to A_P$ *where* $A_P = (S_A, L_A, D_A, F_A, s_{0_A})$ *and where*

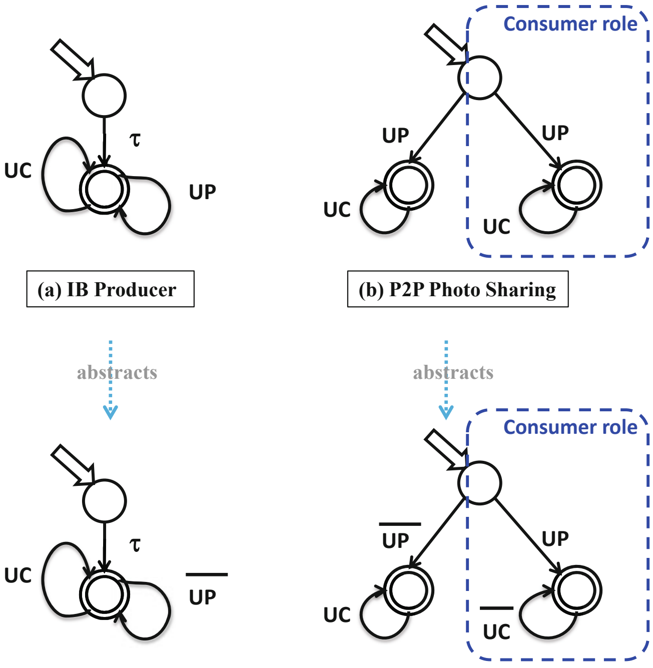


Fig. 13. Abstracted LTSs of the Photo Sharing protocols

$S_A \subseteq S_P$,
$L_A = \{l \in O_{PQ}\} \bigcup \{\tau\}$,
$D_A = \{s_i \xrightarrow{l} s_j \ (or\ s_i \xrightarrow{\bar{l}} s_j) : \exists\ s_k \overset{w}{\Rightarrow} s_n \in D_P \wedge l\ (or\ \bar{l}) = maps_P(w)\}$,
$F_A \subseteq F_P$, *and*
$s_{0_A} = s_{0_P}$.

The above definition applies similarly to Q: *relabels* : $(Q,\ maps_Q,\ O_{PQ}) \rightarrow A_Q$.

In the Photo Sharing scenario, the only label that is not abstracted in the common language is *authenticate* that represents a third party coordination. The IB producer and P2P Photo-Sharing version 1's abstracted LTSs are shown in Figure 13 where the upper part illustrates the protocols on the common language (i.e., common labels without taking into account output and input) while the bottom part of the figure illustrates the protocols on the common language projected on the protocols (i.e., labels where output and inputs are not abstracted). The subsequent step is to check whether the two abstracted protocols share a *complementary coordination policy*, i.e., whether the abstracted protocols may in fact synchronize, which we check over protocol traces as mentioned before.

4.4 Matching Formalization

The formalization described so far is needed to: (1) characterize the protocols and (2) abstract them into protocols on the same alphabet. Then, to establish whether two protocols P and Q can interoperate given their respective abstractions A_P and A_Q based on their common ontology O_{PQ} (i.e., common language) and possibly τs, we need to check that the abstracted protocols A_P and A_Q share complementary coordination policies. To establish this, we use the *functional matching relation* between A_P and A_Q, which succeeds if A_P and A_Q have *a set of pairs of complementary coordination traces*, i.e., at least one pair.

Before going into the definition of the compatibility or functional matching relation, let us provide the one of *complementary coordination policies*. Informally, two coordination policies are complementary if and only if they are the same sequence of actions while having opposite input/output type for all actions. That is, traces t and t' are complementary if and only if: each output action (resp. input) of t has its complementary input action (resp. output) in t' and similarly with switched roles among t' and t. More formally:

Definition 8 (*Complementary Coordination Policies* or *Traces*). *Let:*

- $P = (S_P, L_P, D_P, F_P, s_{0_P})$ *and* $Q = (S_Q, L_Q, D_Q, F_Q, s_{0_Q})$,
- A_P, A_Q *be the abstracted protocols of* P *and* Q *respectively,*
- T_P *and* T_Q *be the set of all the traces of* A_P *and* A_Q*, respectively,*
- $t = l_1, l_2, \ldots, l_n \in T_P$ *and* $t' = l'_1, l'_2, \ldots, l'_m \in T_Q$.

Coordination policies t *and* t' *are complementary coordination policies iff the following conditions hold: discarding the* τ*s,*

(i) for each $l_i \in t : l_i$ *is an output action (input action resp.)* $\exists\ l'_j \in t' : l'_j$ *is an input action (output action resp.);*

(ii) for each $l'_j \in t'$: l'_j is an output action (input action resp.) $\exists\, l_i \in t$: l_i is an input action (output action resp.);

Note that (i) and (ii) above do not take into account the order in which the complementary labels l_i and l'_j are within the traces. Hence, two traces having all complementary labels (skipping the τs) but in different order are considered to be complementary coordination policies (modulo a reordering). Therefore, while doing this check, we store such information that will be used during the mediator synthesis in addition to other information, e.g., the abstraction information.

As said above, we perform the complementary coordination policies check on the abstracted protocols A_P and A_Q, which are expressed in a common language plus τs representing third parties synchronization. We further use the *functional matching relation* to describe the conditions that have to hold in order for two protocols to be compatible. Formally:

Definition 9 (*Compatibility or Functional matching*). *Let:*

- *P and Q protocols,*
- *$relabels$ be a relabeling function,*
- *A_P and A_Q be the abstracted protocols, through $relabels$, of P and Q respectively, and*
- *t_i be a coordination policy of A_P and let t'_i be a coordination policy of A_Q.*

Protocols P and Q have a functional matching (or are compatible) iff there exists a set C of pairs (t_i, t'_i) of complementary coordination policies.

Note that when considering applications that play only the client role, asking for services to a server, the functional matching definition above is slightly modified as follows: instead of checking the existence of a set of pairs of complementary traces, it checks the existence of "a set of pair of traces that result in the *same* trace".

The *functional matching relation* defines necessary conditions that must hold in order for a set of NSs to interoperate through a mediator. In our case, till now, the set is made by two NSs and the matching condition is that they have at least a complementary trace modulo the τs. Such third parties communications (τs) can be just skipped while doing the check, but have to be re-injected while building the mediator. They hence represent information to be stored for the subsequent synthesis.

4.5 Mapping Formalization

Given two protocols P and Q that functionally match, where the set C is made by their pairs of complementary coordination policies, we want to synthesize a mediator M such that the parallel composition $P||M||Q$, allows P and Q to evolve, for their portion C, to their final states. An action of P and Q can belong either to the *common language* or the *third parties language*, i.e., the environment. We build the mediator in such a way that it lets P and Q evolve independently for the portion of the behavior to be exchanged with the environment (denoted by

τ action in the abstracted protocols) until they reach a "synchronization state" from which they can synchronize on complementary actions. We recall that the synchronization cannot be direct since the mediator needs to perform suitable manipulations as for instance actions reordering or translation according to the ontology mapping. An example of translation in the Photo Sharing scenario is $UC = CommentPhoto$ in one protocol and $\overline{UC} = \overline{PhotoComment}$ in the other.

As we said previously, operationally we work on traces instead of working on protocols, hence producing a set of mediating traces for C where we recall that the traces of C's pairs are traces on the abstract protocols A_P and A_Q of P and Q respectively. Then, the mediator protocol AM for C can be easily obtained by merging the mediating traces. AM can be considered an "abstract mediator" since it mediates between abstract protocols. To obtain the corresponding "concrete mediator", we then need to translate each abstract action to its corresponding concrete (sequence of) action(s), i.e., on the languages of P and of Q.

Therefore, a mediator is a protocol that, for each pair $c_{ij} = (c_i, c_j)$ in C, builds a mediating trace m_{ij} such that, for each action (also τ) in c_i and in c_j it always first receive the action and then properly resend it. More formally:

Definition 10 (*Mediator*). *Let:*

- *C be the set of pairs of complementary coordination policies between two abstract protocols A_P and A_Q of protocols P and Q respectively;*
- *O_C be the common language among P and Q;*
- *$(c_i, c_j) \in C$ be a pair of complementary traces where $|c_i| = n$ $|c_i| = m$;*

The mediator M for C is defined as follows:

$$\forall\ (c_i, c_j)\ \exists\ \textit{a mediating trace}\ m_{ij} \in M\ :\ m_{ij} = l_1, l_2, \ldots, l_k\ \wedge\ k = n + m\ \wedge$$
$$\textit{if}\ l_n = \overline{a}\ \wedge\ a \in O_C\ \wedge\ a \in c_i\ \textit{then}\ \exists\ 1 \leq h < n\ :\ l_h = a\ \wedge\ a \in c_j;$$
$$\textit{if}\ l_n = \overline{a}\ \wedge\ a \in O_C\ \wedge\ a \in c_j\ \textit{then}\ \exists\ 1 \leq h < n\ :\ l_h = a\ \wedge\ a \in c_i;$$

The mediator is logically made up of two separate components: M_C and M_T. M_C speaks only the common language and M_T speaks only the third parties language. M_C is a LTS built starting from the common language between P and Q whose aim is to solve the protocol-level mismatches occurring among their dual interactions (complementary sequences of actions) by translating and coordinating between them. M_T, if it exists, is built starting from the third parties language of P and Q and represents the environment. The aim of M_T is to let the protocols evolve, from the initial state or from a state where a previous synchronization is ended, to the states where they can synchronize again.

4.6 Application of the Theory to the Scenario

As already mentioned in Section 4, we assume to have the behavioral specification of the considered Photo Sharing applications, their respective ontologies describing their actions, and the abstraction ontology mapping that defines the common language between IB producer and P2P Photo Sharing. The first step is to *abstract* the protocols exploiting the ontology mapping. Following the theory, the abstracted protocols for the Photo Sharing scenario are illustrated in Figure 13. The second step is the functional matching, i.e., check whether they have some complementary coordination policies. In this scenario, the IB producer is able to simulate the P2P consumer (under complementarity of actions), i.e., right branch of the LTS in Figure 13. The left branch, outside the dashed line, has to be discarded since it is not common with the producer application (while being common with the server of the IB application). Then, the coordination policies that IB producer and P2P consumer share are exactly the consumer's ones.

In this case, only the producer has third parties language actions and then the mediator is made by the part that translates and coordinates the common language and the part that simulates the environment by forwarding from and to it. Hence, with the application of the theory to the scenario, we obtain the connector as shown in Figure 14.

We recall (as already sketched in Section 2.2) that the high level functionalities of the various applications are the following. Taking the producer perspective (1) *authentication* - for the IB producer only -, (2) *upload of photo*, and (3) *download of comments*, while taking the consumer perspective: (i) *download of photo*, and (ii) the *upload of comments.*

The mediator allows the interaction between the two different Photo Sharing applications by (A) manipulating and forwarding the conversations from one protocol to the other and (B) forwarding the interactions between the producer and its server. In the following, we also refer to the Basic Mediator Patterns [66] used to detect and solve the mismatches.

– The IB producer implements the authentication with the action "$\overline{\textit{Authenticate}}$" while the P2P does not include such functionality, i.e., there is no semantically correspondent action in the P2P application (the comple-

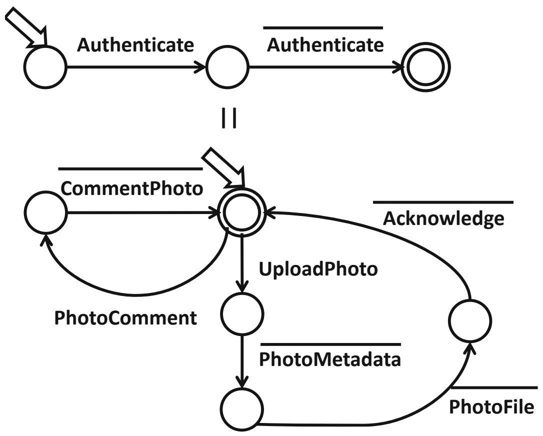

Authenticate
$\overline{\text{Authenticate}}$
||
$\overline{\text{CommentPhoto}}$
$\overline{\text{Acknowledge}}$
UploadPhoto
PhotoComment
$\overline{\text{PhotoMetadata}}$
$\overline{\text{PhotoFile}}$

Fig. 14. Behavioural description of the Mediating Connector for the Photo Sharing example (IB photo producer of Figure 3 a) and P2P Photo Sharing of Figure 2)

mentary action is in the IB server – third parties communication). Then, in this case, the mediator has to forward the interactions from the producer to its server (case (B) above).

- The IB producer implements the upload of photo with the sequence of actions "$\overline{UploadPhoto}$. *Acknowledge*" where the former action sends both photo metadata and file and the latter models the reception of an acknowledgment. The corresponding download of photo implemented by the P2P is the sequence of actions "$PhotoMetadata$. $PhotoFile$". Hence, although the actions are semantically equivalent, they do not synchronize. In order to solve the mismatches among the upload/download of photo, the mediator has to *split* "$\overline{UploadPhoto}$" into "$PhotoMetadata$. $PhotoFile$" and then *produce* the "*Acknowledge*". To detect and solve the mismatches the mediator can respectively leverage on *message splitting pattern* and *message producer pattern* [66]. In this case, the mediator (case (A) above) manipulates and forwards the actions from one protocol to the other.
- The P2P implements the upload of comments with the action "$\overline{PhotoComment}$" while the IB producer implements the respective download of comments with the action "*CommentPhoto*". In order to solve the described mismatch the mediator has to perform a properly translation of "$\overline{PhotoComment}$" into "*CommentPhoto*".

  In order to detect and solve the described signature mismatch, the mediator can use the *message translator pattern* [66]. In this case (case (A) above), the mediator manipulates and forwards the conversations from one protocol to the other.

Note that the building of a connector can be slightly different according to the kind of protocols to be mediated.

If the control of a protocol P is characterized by both send and receive actions, then the mediator will (i) receive an action(s) from P, (ii) properly manipulate it(them), and (iii) send it(them) to the compatible protocol Q of P and vice versa with switched roles between P and Q. Hence the mediator will synchronize with P (Q resp.) to both receive or send messages.

Instead, if the control of protocol P (Q resp.) is only characterized by send actions (i.e., it implements the client role only) then the mediator will only receive actions from P (Q).

5 Related Works

In this chapter we introduced application-layer connectors by referring to both coordinators and mediators. According to these two notions of connector, in this section, we discuss related work in the areas of both automatic coordinator synthesis (Section 5.1) and automatic mediator synthesis (Section 5.2). Indeed, since a mediator can be also seen as a coordinator that enables communication, these works are all related to the automatic mediator synthesis.

5.1 Automatic Synthesis of Coordinators

The architectural approaches to correct and automatic coordinator synthesis presented in Section 3 are related to a large number of other problems that have been considered by researchers over the past two decades. For the sake of brevity we mention below only the works closest to those approaches. The most strictly related approaches are in the *"scheduler synthesis"* research area. In the discrete event domain they appear as *"supervisory control"* or *"discrete controller synthesis"* problem [16,60] addressed by Wonham, Ramadge et al. In very general terms, these works can be seen as an instance of the interoperability problem as (re)phrased in Section 3. However, the application domain of these approaches is sensibly different from the software component domain. Dealing with software components introduces a number of further problematic dimensions to the original synthesis problem. In the scheduler synthesis approaches the possible system executions are modeled as a set of event sequences, and the system specification describes the desired executions. The role of the supervisory controller is to interact with the system in order to meet system specification. The aim of these approach is to restrict the system behavior so that it is contained in a desired behavior, called the *specification*. To do this, the system is constrained to perform events only in strict synchronization with another system, called the *supervisor* (or *controller*). This is achieved by automatically synthesizing a suitable supervisor with respect to the system specification. In contrast to our method, there is one main assumption to deal with deadlocks: in order to automatically synthesize a *supervisor* which avoids deadlocks, they need to consider a specification of the deadlocking behaviors of the base system (i.e., the event sequences that might cause deadlocks). This is a problem because, for large systems, the designers might not know the deadlocking behaviors since they might be unpredictable.

Other works that are related to our approach appear in the *model checking of software components* context in which *compositional reachability analysis* [33] and *automatic assumption generation* [34] techniques are largely used. In [33] Giannakopoulou, Kramer and Cheung described a compositional approach to efficiently perform functional analysis of distributed systems. They validate the behavior of a distributed system with respect to specified safety and liveness properties. The hierarchical software architecture imposed on the system model to be validated allows them to reduce its size. In fact, by exploiting the system hierarchical structure, they are able to check its subsystems against the specified properties. At this point, each subsystem can be minimized in order to be modeled as a single component and the analysis is incrementally carried on. In contrast to our method they are able to minimize the model of the global system by performing efficient analysis. However, the problem faced by their approach is limited to analysis while our technique goes beyond *analyzing* functional properties of a system by also considering the problem of *automatically forcing* the system to exhibit only deadlock-free and specified behaviors. In [34] Giannakopoulou, Pasareanu and Barringer faced a problem that can be seen as an instance of the general problem (re)formulated in Section 3. In the case of these approaches, the treated problem can be formulated as follows: given a component C and a desired behavior B, find an environment E for C in such a way that $E(C) \equiv B$ under an appropriate notion of equivalence. In this approach when model checking a component against a property, the algorithm returns one of the following three results: i) the component satisfies the property for any environment; ii) the component violates the property for any environment; or finally iii) an automatically generated set of assumptions that characterizes exactly those environments in which the component satisfies the property. The difference with our approach is that they automatically synthesize the assumptions that represent the *weakest* environment in which the component satisfies the specified properties. That is, they deal with only two components: i) one actual component and ii) its environment. Moreover, they find an environment in such a way that the specified property is ensured but they do not guarantee the property for any possible environment.

Promising formal techniques for the compositional analysis of component-based design have been developed in [24,55]. The key of these works is the modular-based reasoning that provides a support for the modular checking of behavioral properties. In [24], De Alfaro and Henzinger use an automata-based approach to capture both input assumptions about the order in which the methods of a component are called, and output guarantees about the order in which the component calls external methods. The formalism supports automatic compatibility checks between interface models, and thus constitutes a type system for components interaction. The purpose of this work is different from ours. The authors check that two components have compatible interfaces if a legal environment letting them correctly interact there exists. Each legal environment is an adaptor for the two components. They provide only a consistency check among components interfaces. That is they do not deal with automatic synthesis of component interface adaptors (i.e., automatic synthesis of legal environments). However in [55] De Alfaro, Henzinger, Passerone and Sangiovanni-Vincentelli use a game theoretic approach for checking whether incompatible component interfaces can be made compatible by inserting a converter between them which satisfies specified requirements. This approach is able to automatically synthesize the converter. In contrast to the works described in Section 3, with respect to deadlock-freedom, the specification of the converter's requirements is assumed to be correct. Thus if, e.g., the specification would erroneously introduce deadlocks, they would not be prevented by the converter that it is synthesized in order to be completely compliant to its requirements specification. In other words, a *deadlock preventing* specification of the requirements to be satisfied by the adaptor has to be provided by delegating to the user the non-trivial task of specifying it.

Our research is also related to work in the area of protocol adaptor synthesis developed by Yellin and Strom [88]. The main idea is to modify the interaction mechanisms that are used to glue components together so that compatibility is achieved. This is done by integrating the interaction protocol into components by means of adaptors. However, they are limited to only consider syntactic incompatibilities between the interfaces of components and they do not allow the kind of interaction behavior that our synthesis approach supports. Moreover, they require a formal specification of the adaptor dictating, for example, a mapping function among events of different components. Although requiring this kind of specification enhances applicability of their approach respect to the one described in Section 3, it is in contrast with our need to be as automatic as possible. In fact even if other kinds of techniques to specify the adaptor are possible, providing the adaptor specification requires to know too many implementation details thus missing part of the goals of the work presented in Section 3. However, if we assume to have as input that detailed adaptor specification, our approach can be used to deal with the kind of incompatibilities that Yellin and Strom face in their work. In [7,71], we extended the approach described in Section 3.1 in order to not only restrict the coordinator behavior but also augmenting it in order to consider also such incompatibilities.

In other work from Bracciali, Brogi and Canal [15,20], in the area of component adaptation, it is shown how to automatically generate a concrete adaptor from: (i) a specification of component interfaces, (ii) a partial specification of the components interaction behavior, (iii) a specification of the adaptation in terms of a set of correspondences between actions of different components and (iv) a partial specification of the adaptor. The key result is the setting of a formal foundation for the adaptation of heterogeneous components that may present mismatching interaction behavior. Analogously to the work of Yellin and Strom, although this work provides a fully formal definition of the notion of component adaptor, its application domain is different from our. Since, in specifying a system, we want to maintain a high abstraction level, assuming a specification of the adaptation in terms of a set of correspondences between methods (and their parameters) of two components requires to know many implementation details (about the adaptation) that we do not want to consider in order to synthesize the adaptor.

Concerning the research underpinning the SynthesisRT tool, a related work in synchronous programming is the synchronizing of different clocks. In [23], each input and output port is associated with a periodic clock. Adaptation is performed at the level of each connection between ports using finite buffers. It is sufficient to look at the clocks of two connected ports and to introduce a delay by interposing a *node buffer* between the two ports. In the context of the work described in Section 3.4, adaptation must be performed at the component level by taking into account several dimensions of the specification: the component clock, the interaction protocol, the latency, duration, and controllability of each action. For this reason, introducing delays is not sufficient and, e.g., the reordering or inhibition of actions is also required.

5.2 Automatic Synthesis of Mediators

The automatic synthesis of application-layer mediators presented in Section 4 relates to a wide number of works in the literature within different research areas, beyond the ones discussed in Section 5.1. The theory concentrated on the interoperability problem between heterogeneous protocols within the UbiComp environment.

UbiComp was proposed by Mark Weiser in Nineties [81] [80] as the direction for development of technology in the twenty-first century. But the early basics for this new philosophy were created in 1988 as "tabs, pads and boards" [82]. One of the key principles of UbiComp is to make the computer able to vanish in the background to increase their use making it in an efficient and invisible manner to users. UbiComp suggests the ability for users to enter the environment in a natural way, without being a priori aware of who or what populates it. Furthermore, the user should be able to use the services available using its devices without complex procedures and manual configurations. The currently available technologies and computations have not yet reached the maturity required by the ubiquitous paradigm due to the fact that they are still tied and dependent on the underlying layers although we can qualify them as ubiquitous. The ubiquitous vision fits perfectly with our idea of mediator. Each entity, indeed, maintains its own characteristics (and diversities), being able to communicate and cooperate with the others without having any prior knowledge of them thanks to the support provided by the mediators that masks divergencies making them appear homogeneous.

Interoperability and *mediation* have been investigated in several contexts, among which integration of heterogeneous data sources [85,84], software architecture [32], architectural patterns [18], design patterns [31], patterns of connectors [83,68], Web services [11,22,70,45,38], and algebra to solve mismatches [26] to mention a few.

In particular the interoperability/mediation of protocols have received attention since the early days of networking. Indeed many efforts have been done in several directions including for example formal approaches to *protocol conversion* [19,44,53], and their extension towards reducing the algorithmic complexity of protocol conversion [43].

A work strictly related to the mediators presented in this chapter is, again, the work by Yellin and Strom [88] discussed in Section 5.1. With respect to our mediator synthesis approach, this work prevents to deal with ordering mismatches and different granularity of the languages (one send-many receive and many send-one receive mismatches [66]).

Recently, with the emergence of *Web services* and advocated universal interoperability, the research community has been studying solutions to the *automatic mediation of business processes* [78,77,50,86]. They differ with respect to: (a) a priori exposure of the process models associated with the protocols that are executed by networked resources, (b) knowledge assumed about the protocols run by the interacting parties, (c) matching relationship that is enforced. However, most solutions are discussed informally, making it difficult to assess their respective advantages and drawbacks.

This highlights the needed for a new and formal foundation for mediating connectors from which protocol matching and associated mediation may be rigorously defined and assessed. These relationships should be automatically reasoned upon, thus paving the way for on the fly synthesis of mediating connectors. To the best of our knowledge, such an effort has not been addressed in the Web services and Semantic Web area although proposed algorithms for automated mediation manipulates formally grounded process models.

Within the Web Services research community, a lot of work has been also devoted to *behavioral adaptation* which has been actively studying this problem. Among these works, and related to our, there is [51]. It proposes a *matching approach* based on heuristic algorithms to match services for the adapter generation taking into account both the interfaces and the behavioral descriptions. Our matching is driven by the ontology as described in Section 4 and in [67,36].

Moreover, recently the Web services community has been also investigating how to actually support *service substitution* so as to enable interoperability with different implementations (e.g., due to evolution or provision by different vendors) of a service. While early work has focused on semi-automated, design-time approaches [50,59], latest work concentrates on automated, run-time solutions [25,21]. The work [25] addresses the interoperability problem between services and provide experimentation on real Web2.0 social applications. They propose a technique to dynamically detect and fix interoperability problems based on a catalogue of inconsistencies and their respective adapters. This is similar to our proposal to use *ontology mapping* to discover mismatches and mediator to solve them. Our work differs with respect to theirs because we aim at automatically synthesizing the mediator. Instead, their approach is not fully automatic since although they discover and select mismatches dynamically, the identification of mismatches and of the opportune adapters is made by the engineer.

Our work also closely relates to [21], sharing the exploitation of ontology to reason about interface mapping and the synthesis of mediators according to such mapping. Despite these similarities, our work goes one step further by not being tight to the specific Web service domain.

Our work also closely relates to significant effort from the *semantic Web service* domain and in particular the WSMO (Web Service Modeling Ontology) initiative that defines mediation as a first class entity for Web service modeling towards supporting service composition. The resulting Web service mediation architecture highlights the various mediations levels that are required for systems to interoperate in a highly open network [70]: data level, functional level, and process level. This has in particular led to elicit base patterns for process mediation together with supporting algorithms [22,78].

A lot of work has also been devoted to *connectors* and include a *classification framework* [48], *studies on connectors* [87,41], and *formally grounded* works on connectors. For example, [69] presents an approach for formally specifying connector wrappers as protocol transformations, modularizing them, and reasoning about their properties, with the aim to resolve component mismatches. Another formal work is [28] the authors propose mathematical techniques as foundations to develop architectural design environments that are ADL-independent. Authors of [46] present a formal specification mechanism, by a categorical semantics, for higher order connectors concept that is connectors that take a connector as parameter and deliver another as result. In [10] the authors present a formalization of software connectors. In [17] the authors present an algebra for five basic stateless connectors that are symmetry, synchronization, mutual exclusion, hiding and inaction. They also give the operational, observational and denotational semantics and a complete normal-form axiomatization. The presented connectors can be composed in series and in parallel. A PhD thesis [9] proposes a new connector model, in the distributed component-based context of the SOFA/DCUP project component model. The proposed model allows the description of interactions between components with a semi-automatic generation of the corresponding code.

6 Conclusion and Future Perspectives

Automated and on-the-fly interoperability is a key requirement for heterogeneous protocols within ubiquitous computing environments where networked systems *meet dynamically* and need to *interoperate without a priori knowledge* of each other. Although numerous efforts has been done in many different research areas, such kind of interoperability is still an open challenge.

In Connect, we concentrated on the automatic synthesis of mediators between compatible protocols which enables them to communicate.

We proposed *rigorous techniques* to automatically reason about and compose the behavior of networked systems that aim at fulfilling some goal by connecting to other systems.

The reasoning serves to find a way to achieve communication -if it is possible- and to build the related mediation solution. Our current work put the emphasis on "the elicitation of a way to achieve communication" while it can gain from more practical treatment of similar problems in the literature like the coordinators synthesis or, e.g., converters or adaptors. In particular, we contributed with:

– the design of a comprehensive mediator synthesis process described in [64];
– a set of mediator patterns which represent the building blocks to tackle in a systematic way the protocol mediation. This led us to devise a complete characterization of the protocol mismatches that we are able to solve by our connector synthesis process and to define significant mediator patterns as solution to the classified problems. This is reported in [66] and is revised and extended in [65];
– a formalization of a theory of emerging mediating connectors which includes related automated model-based techniques and tools to support the devised synthesis process. The theory rigorously characterizes: (i) application layer protocols, (ii) their abstraction, (iii) the conditions under which two protocols are functionally matching, (iv) the notion of interoperability between protocols based on the definition of the functional matching relationship, and (v) the mapping, i.e., the synthesis of the mediator behavior needed to achieve protocol interoperability under functional matching. This is illustrated in Section 4 as well as in [67,36,64].

– A combined approach including the theory and a monitoring system towards taking into account also non-functional properties reported in [14].

In the following we discuss *future work perspectives.* The theory of mediators proposed in this chapter (1) clearly defines the interoperability problem, (2) shows the feasibility of the automated reasoning about protocols, i.e., functional matching, and (3) shows the feasibility of the automated synthesis of abstract mediators under certain assumptions. In the future we plan to:

– implement the theory algorithms in order to automatize the mediator generation. In this direction, we are currently working on on-the-fly reasoning about interoperability using ontology-based model checking [12];
– extend the theory of mediators so to have a comprehensive framework for dealing also with middleware layer protocols and data, in addition to application layer protocols. This is currently being investigated [13];
– study run-time techniques towards efficient synthesis;
– scale the synthesis process. The current theory is described considering only two protocols but extending it to an arbitrary number n of protocols seems not to be problematic. The protocol abstraction step developed within the devised process represents a first attempt in this direction by reducing the size of the behavioral models of the NSs to be connected;
– extend the validation of the theory on other real world applications. It would possibly help in tuning the theory, if needed, and in refining the boarders among which the theory works;
– translate the synthesized connector model into an executable artefact that can be deployed and run in the network for actual enactment of the connectors, as studied in the CONNECT project. This also requires devising the runtime architecture of CONNECTors (see Deliverables D1.1 [1] and D1.2 [2] of CONNECT) by investigating the issue of generation of code versus interpretation of the connector model. First results in this direction are in Deliverable D3.2 [3];
– ensure dependability. While preliminary results towards this aim have been described in [14], we aim to take into account both functional interoperability and non-functional interoperability *during* the process. Indeed we would include also the modeling of non-functional aspects, together with their respective matching and mapping reasoning.
– relax some assumptions, towards a dynamic environment, and manage the consequent uncertainty. For example, we aim at integrating with complementary works ongoing within the CONNECT project so as to develop an overall framework enabling the dynamic synthesis of emergent connectors among networked systems. Instances of complementary works are (i) learning techniques to dynamically discover the protocols (instead of assuming them given) that are run in the environment; (ii) data-level interoperability (instead of assuming the ontology given) to elicit the data mappings. This may rise the problem of dealing with partial or erroneous specifications.

References

1. CONNECT consortium. CONNECT Deliverable D1.1: Initial Connect Architecture. FET IP CONNECT EU project, FP7 grant agreement number 231167, `http://www.connect-forever.eu/`
2. CONNECT consortium. CONNECT Deliverable D1.2: Intermediate Connect Architecture. FET IP CONNECT EU project, FP7 grant agreement number 231167, `http://www.connect-forever.eu/`
3. CONNECT consortium. CONNECT Deliverable D3.2: Reasoning about and Harmonizing the Interaction Behavior of Networked Systems at Application- and Middleware- Layer. FET IP CONNECT EU project, FP7 grant agreement number 231167, `http://www.connect-forever.eu/`
4. ITU Telecommunication Standardisation sector, ITU-T reccomendation Z.120. Message Sequence Charts (MSC 1996), Geneva (1996)
5. Arnold, A.: Finite Transition Systems. International Series in Computer Science. Prentice Hall International, UK (1989)
6. Autili, M., Inverardi, P., Navarra, A., Tivoli, M.: Synthesis: A tool for automatically assembling correct and distributed component-based systems. In: 29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, pp. 784–787. IEEE Computer Society, Los Alamitos (2007), DOI REF: `http://doi.ieeecomputersociety.org/10.1109/ICSE.2007.84`
7. Autili, M., Inverardi, P., Tivoli, M., Garlan, D.: Synthesis of "correct" adaptors for protocol enhancement in component based systems. In: Proceedings of the 1st International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2004) at FSE 2004, pp. 79–86 (2004)
8. Autili, M., Mostarda, L., Navarra, A., Tivoli, M.: Synthesis of decentralized and concurrent adaptors for correctly assembling distributed component-based systems. Journal of Systems and Software 81(12), 2210–2236 (2008)
9. Balek, D.: Connectors in Software Architectures. PhD thesis, Charles University (May 2002)
10. Barbosa, M.A., Barbosa, L.S.: Specifying software connectors. In: Liu, Z., Araki, K. (eds.) ICTAC 2004. LNCS, vol. 3407, pp. 52–67. Springer, Heidelberg (2005)
11. Benatallah, B., Casati, F., Grigori, D., Nezhad, H.R.M., Toumani, F.: Developing adapters for web services integration. In: Pastor, Ó., Falcão e Cunha, J. (eds.) CAiSE 2005. LNCS, vol. 3520, pp. 415–429. Springer, Heidelberg (2005)
12. Bennaceur, A., Issarny, V., Spalazzese, R.: On-the-fly reasoning about interoperability using ontology-based model checking. Technical Report, INRIA Rocquencourt, Paris (January 2011)
13. Bennaceur, A., Spalazzese, R., Inverardi, P., Issarny, V., Georgantas, N., Saadi, R.: Model-based mediators for dynamic-adaptive connectors. Technical report, INRIA Paris-Rocquencourt, France (2011)
14. Bertolino, A., Inverardi, P., Issarny, V., Sabetta, A., Spalazzese, R.: On-the-fly interoperability through automated mediator synthesis and monitoring. In: Margaria, T., Steffen, B. (eds.) ISoLA 2010, Part II. LNCS, vol. 6416, pp. 251–262. Springer, Heidelberg (2010)
15. Bracciali, A., Brogi, A., Canal, C.: A formal approach to component adaptation. J. Syst. Softw. 74 (January 2005)
16. Brandin, B., Wonham, W.: Supervisory control of timed discrete-event systems. IEEE Transactions on Automatic Control 39(2) (1994)

17. Bruni, R., Lanese, I., Montanari, U.: A basic algebra of stateless connectors. Theor. Comput. Sci. 366(1), 98–120 (2006)
18. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern-Oriented Software Architecture. A System of Patterns, vol. 1. Wiley, Chichester (1996)
19. Calvert, K.L., Lam, S.S.: Formal methods for protocol conversion. IEEE Journal on Selected Areas in Communications 8(1), 127–142 (1990)
20. Canal, C., Poizat, P., Salaün, G.: Model-based adaptation of behavioral mismatching components. IEEE Trans. Software Eng. 34(4), 546–563 (2008)
21. Cavallaro, L., Nitto, E.D., Pradella, M.: An automatic approach to enable replacement of conversational services. In: Baresi, L., Chi, C.-H., Suzuki, J. (eds.) ICSOC-ServiceWave 2009. LNCS, vol. 5900, pp. 159–174. Springer, Heidelberg (2009)
22. Cimpian, E., Mocan, A.: WSMX process mediation based on choreographies. In: Bussler, C., Haller, A. (eds.) BPM 2005. LNCS, vol. 3812, pp. 130–143. Springer, Heidelberg (2006)
23. Cohen, A., Duranton, M., Eisenbeis, C., Pagetti, C., Plateau, F., Pouzet, M.: Synchronization of periodic clocks. In: Proc. of the 5th EMSOFT (2005)
24. de Alfaro, L., Henzinger, T.A.: Interface automata. In: Proceedings of the 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering. ESEC/FSE, vol. 9, pp. 109–120 (2001)
25. Denaro, G., Pezzé, M., Tosi, D.: Ensuring interoperable service-oriented systems through engineered self-healing. In: Proceedings of ESEC/FSE 2009. ACM Press, New York (2009)
26. Dumas, M., Spork, M., Wang, K.: Adapt or perish: Algebra and visual notation for service interface adaptation. In: Dustdar, S., Fiadeiro, J.L., Sheth, A.P. (eds.) BPM 2006. LNCS, vol. 4102, pp. 65–80. Springer, Heidelberg (2006)
27. Feiler, P., Gabriel, R.P., Goodenough, J., Lingerand, R., Longstaff, T., Kazman, R., Klein, M., Northrop, L., Schmidt, D., Sullivan, K., Wallnau, K.: Ultra-Large-Scale Systems: The Software Challenge of the Future (2006)
28. Fiadeiro, J.L., Lopes, A., Wermelinger, M.: Theory and practice of software architectures. In: Tutorial at the 16th IEEE Conference on Automated Software Engineering, San Diego, CA, USA, November 26-29 (2001)
29. Finkel, A.: The minimal coverability graph for Petri nets. In: Rozenberg, G. (ed.) APN 1993. LNCS, vol. 674. Springer, Heidelberg (1993)
30. Blair, G., et al.: Introduction to Interoperability. In: Bernardo, M., Issarny, V. (eds.) SFM 2011. LNCS, vol. 6659, pp. 350–392. Springer, Heidelberg (2011)
31. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Resusable Object-Oriented Software. Addison-Wesley Professional, Reading (1995)
32. Garlan, D., Shaw, M.: An introduction to software architecture. Technical Report CMU-CS-94-166, Carnegie Mellon University (January 1994)
33. Giannakopoulou, D., Kramer, J., Cheung, S.C.: Behaviour analysis of distributed systems using the tracta approach. Automated Software Engg. 6, 7–35 (1999)
34. Giannakopoulou, D., Păsăreanu, C.S., Barringer, H.: Component verification with automatically generated assumptions. Automated Software Engg. 12, 297–320 (2005)
35. Intrigila, B., Inverardi, P., Zilli, M.V.: A comprehensive setting for matching and unification over iterative terms. Fundam. Inform. 39(3), 273–304 (1999)
36. Inverardi, P., Issarny, V., Spalazzese, R.: A theory of mediators for eternal connectors. In: Margaria, T., Steffen, B. (eds.) ISoLA 2010, Part II. LNCS, vol. 6416, pp. 236–250. Springer, Heidelberg (2010)

37. Inverardi, P., Nesi, M.: Deciding observational congruence of finite-state ccs expressions by rewriting. Theor. Comput. Sci. 139(1-2), 315–354 (1995)
38. Jiang, F., Fan, Y., Zhang, X.: Rule-based automatic generation of mediator patterns for service composition mismatches. In: Proceedings of the 2008, The 3rd International Conference on Grid and Pervasive Computing - Workshops, pp. 3–8. IEEE Computer Society, Washington, DC, USA (2008)
39. Kalfoglou, Y., Schorlemmer, M.: Ontology mapping: the state of the art. Knowl. Eng. Rev. 18(1), 1–31 (2003)
40. Kalfoglou, Y., Schorlemmer, M.: Ontology mapping: The state of the art. In: Kalfoglou, Y., Schorlemmer, M., Sheth, A., Staab, S., Uschold, M. (eds.) Semantic Interoperability and Integration, Dagstuhl, Germany. Dagstuhl Seminar Proceedings, vol. 04391. IBFI, Schloss Dagstuhl, Germany (2005)
41. Kell, S.: Rethinking software connectors. In: SYANCO 2007: International Workshop on Synthesis and Analysis of Component Connectors, pp. 1–12. ACM, New York (2007)
42. Keller, R.M.: Formal verification of parallel programs. Commun. ACM 19(7), 371–384 (1976)
43. Kumar, R., Nelvagal, S., Marcus, S.I.: A discrete event systems approach for protocol conversion. Discrete Event Dynamic Systems 7(3) (1997)
44. Lam, S.S.: Correction to "protocol conversion". IEEE Trans. Software Eng. 14(9), 1376 (1988)
45. Li, X., Fan, Y., Wang, J., Wang, L., Jiang, F.: A pattern-based approach to development of service mediators for protocol mediation. In: Proceedings of WICSA 2008, pp. 137–146. IEEE Computer Society, Los Alamitos (2008)
46. Lopes, A., Wermelinger, M., Fiadeiro, J.L.: Higher-order architectural connectors. ACM Trans. Softw. Eng. Methodol. 12(1), 64–104 (2003)
47. Magee, J., Kramer, J.: Concurrency: State models and Java programs. Wiley, Chichester (2006)
48. Mehta, N.R., Medvidovic, N., Phadke, S.: Towards a taxonomy of software connectors. In: ICSE 2000: Proceedings of the 22nd International Conference on Software Engineering, pp. 178–187. ACM Press, New York (2000)
49. Milner, R.: Communication and Concurrency. Prentice Hall, New York (1989)
50. Motahari Nezhad, H.R., Benatallah, B., Martens, A., Curbera, F., Casati, F.: Semi-automated adaptation of service interactions. In: WWW 2007: Proceedings of the 16th International Conference on World Wide Web, pp. 993–1002. ACM, New York (2007)
51. Motahari Nezhad, H.R., Xu, G.Y., Benatallah, B.: Protocol-aware matching of web service interfaces for adapter development. In: Proceedings of the 19th International Conference on World Wide Web, WWW 2010, pp. 731–740. ACM, New York (2010)
52. Murata, T.: Petri nets: Properties, analysis and applications. Proceedings of the IEEE 77(4) (1989)
53. Okumura, K.: A formal protocol conversion method. In: SIGCOMM, pp. 30–37 (1986)
54. Oreizy, P., Medvidovic, N., Taylor, R.N.: Architecture-based runtime software evolution. In: Proceedings of the 20th International Conference on Software Engineering, ICSE 1998, pp. 177–186 (1998)
55. Passerone, R., de Alfaro, L., Henzinger, T.A., Sangiovanni-Vincentelli, A.L.: Convertibility verification and converter synthesis: two faces of the same coin. In: Proceedings of the 2002 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2002, pp. 132–139 (2002)

56. Pelliccione, P., Inverardi, P., Muccini, H.: Charmy: A framework for designing and verifying architectural specifications. IEEE Trans. Softw. Eng. 35, 325–346 (2009)
57. Pelliccione, P., Tivoli, M., Bucchiarone, A., Polini, A.: An architectural approach to the correct and automatic assembly of evolving component-based systems. Journal of Systems and Software 81(12), 2237–2251 (2008)
58. Plotkin, G.D.: A note on inductive generalization. Machine Intelligence 5, 153–163 (1970)
59. Ponnekanti, S., Fox, A.: Interoperability among independently evolving web services. In: Jacobsen, H.-A. (ed.) Middleware 2004. LNCS, vol. 3231, pp. 331–351. Springer, Heidelberg (2004)
60. Ramadge, P., Wonham, W.: Supervisory control of a class of discrete event processes. Siam J. Control and Optimization 25(1) (1987)
61. Ramadge, P., Wonham, W.: The control of discrete event systems. Proceedings of the IEEE 1(77) (1989)
62. Reynolds, J.: Transformational systems and the algebraic structure of atomic formulas machine intelligence, vol. 5, pp. 135–151. Edinburgh University Press, USA (1970)
63. Roscoe, A.W., Hoare, C.A.R., Bird, R.: The Theory and Practice of Concurrency. Prentice Hall PTR, Upper Saddle River (1997)
64. Spalazzese, R.: A Theory of Mediating Connectors to achieve Interoperability. PhD thesis, University of L'Aquila (April 2011)
65. Spalazzese, R., Inverardi, P.: Components interoperability through mediating connector pattern. In: WCSI 2010, arXiv:1010.2337. EPTCS, vol. 37, pp. 27–41 (2010)
66. Spalazzese, R., Inverardi, P.: Mediating connector patterns for components interoperability. In: Babar, M.A., Gorton, I. (eds.) ECSA 2010. LNCS, vol. 6285, pp. 335–343. Springer, Heidelberg (2010)
67. Spalazzese, R., Inverardi, P., Issarny, V.: Towards a formalization of mediating connectors for on the fly interoperability. In: Proceedings of the Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture (WICSA/ECSA 2009), pp. 345–348 (2009)
68. Spitznagel, B.: Compositional Transformation of Software Connectors. PhD thesis, Carnegie Mellon University (May 2004)
69. Spitznagel, B., Garlan, D.: A compositional formalization of connector wrappers. In: ICSE, pp. 374–384 (2003)
70. Stollberg, M., Cimpian, E., Mocan, A., Fensel, D.: A semantic web mediation architecture. In: Proceedings of the 1st Canadian Semantic Web Working Symposium (CSWWS 2006). Springer, Heidelberg (2006)
71. Tivoli, M., Autili, M.: Synthesis, a tool for synthesizing correct and protocol-enhanced adaptors. RSTI - L'objet, Coordination and Adaptation Techniques 12(1), 77–103 (2006)
72. Tivoli, M., Fradet, P., Girault, A., Gößler, G.: Adaptor synthesis for real-time components. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 185–200. Springer, Heidelberg (2007)
73. Tivoli, M., Inverardi, P.: Failure-free coordinators synthesis for component-based architectures. Science of Computer Programming 71(3), 181–212 (2008)
74. Uchitel, S., Kramer, J.: A workbench for synthesising behaviour models from scenarios. In: Proceeding of the 23rd IEEE International Conference on Software Engineering (ICSE 2001) (2001)
75. Uchitel, S., Kramer, J., Magee, J.: Detecting implied scenarios in message sequence chart specifications. In: ACM Proceedings of the Joint 8th ESEC and 9th FSE (2001)

76. Issarny, V., et al.: Middleware-layer Connector Synthesis. In: Bernardo, M., Issarny, V. (eds.) SFM 2011. LNCS, vol. 6659, pp. 350–392. Springer, Heidelberg (2011)
77. Vaculín, R., Neruda, R., Sycara, K.P.: An agent for asymmetric process mediation in open environments. In: Kowalczyk, R., Huhns, M.N., Klusch, M., Maamar, Z., Vo, Q.B. (eds.) SOCASE. LNCS, vol. 5006, pp. 104–117. Springer, Heidelberg (2008)
78. Vaculín, R., Sycara, K.: Towards automatic mediation of OWL-S process models. In: IEEE International Conference on Web Services, vol. 0, pp. 1032–1039 (2007)
79. Watt, S.M.: Algebraic generalization. SIGSAM Bull. 39(3), 93–94 (2005)
80. Weiser, M.: The computer for the 21st century. Scientific American (September 1991)
81. Weiser, M.: Hot Topics: Ubiquitous Computing. IEEE Computer (October 1993)
82. Weiser, M.: Ubiquitous computing (1996), `http://sandbox.xerox.com/ubicomp/`
83. Wermelinger, M., Fiadeiro, J.L.: Connectors for mobile programs. IEEE Trans. Softw. Eng. 24(5), 331–341 (1998)
84. Wiederhold, G.: Mediators in the architecture of future information systems. IEEE Computer 25, 38–49 (1992)
85. Wiederhold, G., Genesereth, M.: The conceptual basis for mediation services. IEEE Expert: Intelligent Systems and Their Applications 12(5), 38–47 (1997)
86. Williams, S.K., Battle, S.A., Cuadrado, J.E.: Protocol mediation for adaptation in semantic web services. In: Sure, Y., Domingue, J. (eds.) ESWC 2006. LNCS, vol. 4011, pp. 635–649. Springer, Heidelberg (2006)
87. Woollard, D., Medvidovic, N.: High performance software architectures: A connector-oriented approach. In: Proceedings of the Institute for Software Research Graduate Research Symposium, Irvine, California (June 2006)
88. Yellin, D.M., Strom, R.E.: Protocol specifications and component adaptors. ACM Trans. Program. Lang. Syst. 19 (1997)

Context Synthesis

Dimitra Giannakopoulou[2] and Corina S. Păsăreanu[1]

[1] Carnegie Mellon Silicon Valley/
[2] NASA Ames Research Center,
Moffett Field, CA 94035, USA
{dimitra.giannakopoulou,corina.s.pasareanu}@nasa.gov

Abstract. With the advent of component-based and distributed software development, service-oriented computing, and other such concepts, components are no longer viewed as parts of specific systems, but rather as open systems that can be reused, or connected dynamically, in a variety of environments to form larger systems. Reasoning about components as open systems is different from reasoning about closed systems, since property satisfaction may depend on the context in which a component may be introduced.

Component interfaces are an important feature of open sytems, since interfaces summarize the expectations that a component has from the contexts in which it gets introduced. Traditionally, component interfaces have been of a purely syntactic form, including information about the services/methods that can be invoked on the component, and their signatures, meaning the numbers and types of arguments and their return values. However, there is a recognized need for richer interfaces that capture additional aspects of a component. For example, interfaces may characterize legal sequences of invocations to component services.

Generating compact and yet useful component interfaces is a challenging task to perform manually. Over the last decade, several approaches have been developed for performing *context synthesis*, i.e., generating component interfaces automatically. This tutorial mostly reviews such techniques developed by the authors, but also discusses alternative techniques for context synthesis.

1 Introduction

With the advent of component-based and distributed software development, service-oriented computing, and other such concepts, components are no longer viewed as parts of specific systems, but rather as open systems that can be reused, or connected dynamically, in a variety of environments to form larger systems. Reasoning about components as open systems is different from reasoning about closed systems, since property satisfaction may depend on the context in which a component may be introduced. As a result, a component satisfies or violates a property only when the property is satisfied or violated by the component irrespective of context. For all other cases, meaningful analysis would consist of synthesizing a characterization of all contexts in which the component satisfies the desired property. We refer to such analysis as "context synthesis", and the result of the synthesis a "component interface".

Component interfaces therefore summarize the expectations that a component has from the contexts in which it gets introduced. Traditionally, component interfaces have been of a purely syntactic form, including information about the services/methods that can be invoked on the component, and their signatures, meaning the numbers and types of arguments and their return values. However, there is a recognized need for richer interfaces that capture additional aspects of a component. For example, "temporal" interfaces [3], which are the focus of this tutorial, describe legal sequences of service invocations or method calls to a component. The purpose is to document (for clients of a component) what sequences of calls could lead to undesirable component states and should therefore be avoided.

Generating compact and yet useful component interfaces is a challenging task to perform manually. Over the last decade, several approaches have been developed for generating component interfaces automatically. This tutorial reviews techniques for interface generation of components with respect to safety properties. We discuss in depth some techniques developed by the authors, but also present and discuss alternative techniques, and provide references to some new trends in this research area.

Context synthesis is closely related to compositional reasoning methods for model checking. Compositional verification presents a "divide and conquer" approach to the state-explosion problem [9] associated with model checking. It decomposes the properties of the system into local properties of the system components. Each component is checked separately against its local properties; the combination of these simpler checks guarantees the correctness of the global property on the entire system.

Analysis of components in isolation for compositional verification will often return results that are not meaningful. The reason is again that components usually rely on some features of the environments in which they are introduced. One therefore needs to incorporate some knowledge of the contexts in which the components are expected to operate correctly. Assume-guarantee reasoning [19,24] addresses this issue by making explicit use of assumptions in component verification. Assumptions are akin to interfaces and they document expectations of a component from its environment in order to fulfill its guarantees. Assume-guarantee rules are then used to merge the results of individual component verification steps for verification of system-level properties.

The fundamental difference between an interface and an assumption in the context of reasoning about safety properties, is that an interface summarizes the component with respect to *all* the possible environments in which the component may be introduced. On the other hand, an assumption serves as a potentially imprecise interface that only needs to reflect interactions with a *specific* environment, representing the rest of the components in the analyzed system. We note that in the context of compositional verification, all the components that participate in the verification problem are known and available. As a result, context synthesis takes the form of assumption generation in compositional verification, and can be performed more efficiently than interface generation due to the availability of an actual component environment.

The rest of this paper is organized as follows. We provide background for our work in Section 2, followed by a characterization of precise (safe and permissive) component interfaces in the context of safety property checking for finite state systems in Section 3. In Section 4, we present several algorithms for automated interface generation. A construction of what we call the "weakest assumption", corresponding to a precise component interface, is presented first. We then present an alternative approach that uses the L* learning algorithm to compute component interfaces in an iterative manner. In Section 5 we address the problem of generating interfaces for infinite state components. Context synthesis is subsequently discussed in the context of compositional verification in Section 6. Finally, we discuss implementation and applicability of these techniques and present some open research topics in this domain in Section 7.

2 Background

In this section we introduce labeled transition systems (LTSs) [20], the formalism that we use to model components. We present the definition of traces and parallel composition for LTSs and also present how safety properties are checked. We then introduce assume-guarantee reasoning and the L* algorithm that we use to automatically synthesize interfaces and assumptions.

2.1 Labeled Transition Systems (LTSs)

Let $\mathcal{A}ct$ be the universal set of observable actions and let τ denote a local *unobservable* action. Let π denote a special *error state*, which models safety violations;

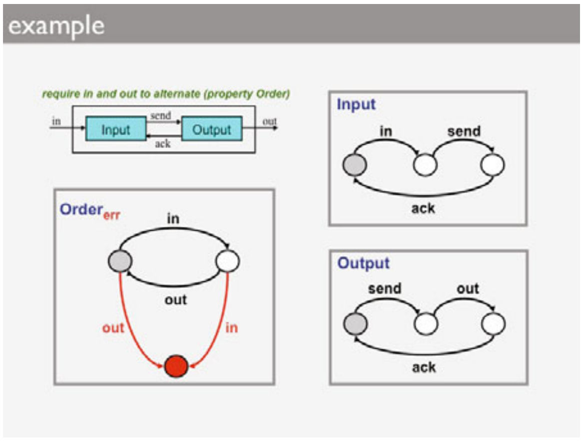


Fig. 1. Example

the error state has no outgoing transitions. Formally, an LTS M is a four-tuple $\langle Q, \alpha M, \delta, q_0 \rangle$ where:

- Q is a finite non-empty set of states
- $\alpha M \subseteq \mathcal{A}ct$ is a set of observable actions called the *alphabet* of M
- $\delta \subseteq Q \times (\alpha M \cup \{\tau\}) \times Q$ is a transition relation
- $q_0 \in Q$ is the initial state

Let Π denote the LTS $\langle \{\pi\}, \mathcal{A}ct, \emptyset, \pi \rangle$. An LTS $M = \langle Q, \alpha M, \delta, q_0 \rangle$ is *non-deterministic* if it contains τ-transitions or if there exists $(q, a, q'), (q, a, q'') \in \delta$ such that $q' \neq q''$. Otherwise, M is *deterministic*.

2.2 Traces

A *trace* t of LTS M is a finite sequence of observable actions that label the transitions that M can perform starting from its initial state (ignoring the unobservable τ-transitions). We sometimes denote by t both a trace and its *trace LTS*. For a trace t of length n, its trace LTS $lts(t)$ consists of $n+1$ states, such that there is a transition between states i and $i+1$ on the i^{th} action in trace t, for $1 \leq i \leq n$. The set of all traces of an LTS M is the language of M,denoted $\mathcal{L}(M)$; $errTr(M)$ denotes the set of traces that lead to π, which are called the *error traces* of M.

For $\Sigma \subseteq \mathcal{A}ct$, we use $t \uparrow \Sigma$ to denote the trace obtained by removing from t all occurrences of actions $a \notin \Sigma$. Similarly, $M \uparrow \Sigma$ is defined to be an LTS over alphabet Σ which is obtained from M by renaming to τ all the transitions labeled with actions that are not in Σ.

2.3 Parallel Composition

Let $M = \langle Q, \alpha M, \delta, q_0 \rangle$ and $M' = \langle Q', \alpha M', \delta', q_0' \rangle$. M *transits* into M' with action a, written $M \xrightarrow{a} M'$, if and only if $(q_0, a, q_0') \in \delta$ and either $Q = Q'$, $\alpha M = \alpha M'$, and $\delta = \delta'$ for $q_0' \neq \pi$, or, in the special case where $q_0' = \pi$, $M' = \Pi$.

The parallel composition operator $\|$ is a commutative and associative operator that combines the behavior of two components by synchronizing the common actions and interleaving the remaining actions.

Formally, let $M_1 = \langle Q^1, \alpha M_1, \delta^1, q_0^1 \rangle$ and $M_2 = \langle Q^2, \alpha M_2, \delta^2, q_0^2 \rangle$ be two LTSs. If $M_1 = \Pi$ or $M_2 = \Pi$, then $M_1 \parallel M_2 = \Pi$. Otherwise, $M_1 \parallel M_2$ is an LTS $M = \langle Q, \alpha M, \delta, q_0 \rangle$, where $Q = Q^1 \times Q^2$, $q_0 = (q_0^1, q_0^2)$, $\alpha M = \alpha M_1 \cup \alpha M_2$, and δ is defined as follows, where a is either an observable action or τ:

$$\frac{M_1 \xrightarrow{a} M_1',\ a \notin \alpha M_2}{M_1 \parallel M_2 \xrightarrow{a} M_1' \parallel M_2} \qquad \frac{M_2 \xrightarrow{a} M_2',\ a \notin \alpha M_1}{M_1 \parallel M_2 \xrightarrow{a} M_1 \parallel M_2'}$$

$$\frac{M_1 \xrightarrow{a} M_1',\ M_2 \xrightarrow{a} M_2',\ a \neq \tau}{M_1 \parallel M_2 \xrightarrow{a} M_1' \parallel M_2'}$$

2.4 Safety Properties

In our context, properties are modeled as *safety LTS*'s. A *safety LTS* is a deterministic LTS that contains no π states. A *safety property* is specified as a safety LTS P, whose language $\mathcal{L}(P)$ defines the set of acceptable behaviors over αP.

For an LTS M and a safety LTS P such that $\alpha P \subseteq \alpha M$, we say that M satisfies P, denoted $M \models P$, if and only if $\forall t \in \mathcal{L}(M) : (t \uparrow \alpha P) \in \mathcal{L}(P)$.

When checking a property P, an *error LTS* denoted P_{err} is created, which traps possible violations with the π state. Formally, the error LTS of a property $P = \langle Q, \alpha P, \delta, q_0 \rangle$ is $P_{err} = \langle Q \cup \{\pi\}, \alpha P_{err}, \delta', q_0 \rangle$, where $\alpha P_{err} = \alpha P$ and

$$\delta' = \delta \cup \{(q, a, \pi) \mid q \in Q, a \in \alpha P, \mathit{and} \nexists q' \in Q : (q, a, q') \in \delta\}$$

Note that the error LTS is *complete* (except for π), meaning each state other than the error state has outgoing transitions for every action in its alphabet. Also note that the error traces of P_{err} define the language of P's complement.

As an example, consider the communication channel in Figure 1. It consists of an `Input` and an `Output` component; grey states are initial states. Actions `send` and `ack` are common to the alphabets of the two components and will be synchronized when the LTSs are composed. Property `Order` states that `inputs` and `outputs` come in matched pairs with `input` always preceding `output`. The error state is colored red.

2.5 Assume-Guarantee Reasoning

Concurrent software is inherently difficult to analyze. The problem of reaching a specific global state in a system with N finite-state components can be shown to be PSPACE-complete in N. What this means in practice is that the number of states in a concurrent system may in the worst case be exponential in the size of the components of the system.

One approach to dealing with this state explosion problem is compositional or local reasoning. In essence, the goal of compositional reasoning is to replace the analysis over the global state space with localized analyses, which consider each component by itself, together with a small abstraction of the environment of that component. The intuition behind this principle is that many systems can be viewed as "loosely coupled" collections of components; that is, the proportion of the behavior of one component behavior which influences that of another is small. Hence, there should be an advantage to doing localized reasoning.

Assume-guarantee reasoning [19,24] is a compositional verification technique that uses assume-guarantee rules for verification. Assume-guarantee rules are proof rules that show how, by performing local verification steps on individual components of the system, one can safely deduce properties that refer to the entire system. The local verification steps usually involve some abstraction of the environment of each component, named *assumption.*

In the assume-guarantee reasoning paradigm, formulas are triples of the type $\langle A \rangle\ M\ \langle P \rangle$, where M is a component, P is a property, and A is an assumption about M's environment. The formula is true if whenever M is part of a system satisfying A, then the system must also guarantee P, *i.e.*, $\forall E$, $E \parallel M \models A$ implies $E \parallel M \models P$. For LTS M and safety LTSs A and P, checking $\langle A\rangle\ M\ \langle P\rangle$ reduces to checking if state π is reachable in $A \parallel M \parallel P_{err}$. Note that when $\alpha P \subseteq \alpha A \cup \alpha M$, this is equivalent to $A \parallel M \models P$. Also note that we assume that M contains no π states.

The simplest assume guarantee rule is for checking a safety propery P on a system with two components M_1 and M_2.
Rule ASym

$$\frac{\begin{array}{l}1 : \langle A\rangle\ M_1\ \langle P\rangle \\ 2 : \langle true\rangle\ M_2\ \langle A\rangle\end{array}}{\langle true\rangle\ M_1 \parallel M_2\ \langle P\rangle}$$

In this rule, A denotes an assumption about the environment of M_1. Note that the rule is not symmetric in its use of the two components, and does not support circularity. Despite its simplicity, experience with applying compositional verification has shown this rule to be most useful in the context of checking safety properties. This rule can be extended for multiple components. Several other rules have also been defined in the literature. We have experimented with several rules in practice [23].

Weakest Assumption. For a given LTS component M and safety property P there is a natural notion of the *weakest assumption* A_w, such that $\langle A_w\rangle\ M\ \langle P\rangle$ holds. A_w characterizes all the possible environments E under which the property holds, *i.e.*$\forall E : M \parallel E \models P$ if and only if $E \models A_w$.

2.6 The L* Algorithm

L* is a learning algorithm that was developed by Angluin [2] and later improved by Rivest and Schapire [25]. L* learns an unknown regular language and produces a DFA that accepts it. Let U be an unknown regular language over some alphabet Σ. In order to learn U, L* needs to interact with a *Minimally Adequate Teacher*, from now on called *Teacher*. A Teacher must be able to correctly answer two types of questions from L*. The first type is a *membership query*, consisting of a string $\sigma \in \Sigma^*$; the answer is *true* if $\sigma \in U$, and *false* otherwise. The second type of question is a *conjecture*, i.e., a candidate DFA C whose language the algorithm believes to be identical to U. The answer is *true* if $\mathcal{L}(C) = U$. Otherwise the Teacher returns a counterexample, which is a string σ in the symmetric difference of $\mathcal{L}(C)$ and U.

At a higher level, L* creates a table where it incrementally records whether strings in Σ^* belong to U. It does this by making membership queries to the Teacher. At various stages L* decides to make a conjecture. It constructs a candidate automaton C based on the information contained in the table and asks the Teacher whether the conjecture is correct. If it is, the algorithm terminates. Otherwise, L* uses the counterexample returned by the Teacher to extend the table with strings that witness differences between $\mathcal{L}(C)$ and U.

Characteristics of L.* L* depends on the correctness of the Teacher in order to provide a number of guarantees. More specifically, L* is guaranteed to terminate with a minimal automaton for the unknown language U. Moreover, each candidate DFA C that L* constructs is smallest, in the sense that any other DFA consistent with the information provided to L* has at least as many states as C. This characteristic of L* makes it particularly attractive in the context of learning interfaces or assumptions, since in the frameworks that we describe, the candidates provided by L* are combined with component models in model checking steps. Smaller state machines typically result in easier model checking problems. The conjectures made by L* strictly increase in size; each conjecture is smaller than the next one, and all incorrect conjectures are smaller than the minimal automaton for language U. Therefore, if that minimal automaton has n states, L* makes at most $n-1$ incorrect conjectures. The number of membership queries made by L* is $\mathcal{O}(kn^2 + n \log m)$, where k is the size of the alphabet of U, n is the number of states in the minimal DFA for U, and m is the length of the longest counterexample returned when a conjecture is made.

3 Component Interfaces

An interface characterizes the expectations that a component has from its environment. As discussed in the introduction, there is a recognized need for extending component interfaces beyond their traditional, purely syntactic form. In this tutorial paper, we focus on interfaces that describe legal sequences of service invocations or method calls to a component. For example, an interface may describe the fact that closing a file before opening it is undesirable because an exception will be thrown. An ideal interface should precisely represent the component in all its intended usages. In other words, it should include all the good interactions, and exclude all problematic interactions. We describe this formally in the following.

Let Σ be the set of interaction points of an LTS M, where $\Sigma \subseteq \alpha M$. A word over Σ is considered *legal* if its execution cannot lead M to an error state, and is considered legal otherwise. Accordingly, we define two additional language sets for any LTS M. We use the term $\mathcal{L}_{illegal}(M) = \mathit{errTr}(M)$ to refer to the set of illegal executions of LTS M. The set of legal executions is defined as $\mathcal{L}_{legal}(M) = \Sigma^* \setminus \mathcal{L}_{illegal}(M)$. Note that we slightly abuse the term "executions" in this context, since the set of legal executions may contain words that cannot execute to completion in M, meaning that they do not correspond to traces in $\mathcal{L}(M)$. The reason why it is desirable to consider such words in the set of legal executions is that such words should never be disallowed in the behavior of M's environment since they can never be executed in the context of M, and could therefore never lead M to an error state.

Let us now assume that a component is represented as an LTS M, with Σ being the set of its interaction points with the environment. Assume also that an interface A is represented as an LTS over Σ. Then A is a precise interface for M if it satisfies two conditions:

1. **Safe.** Interface A is safe for M iff $\mathcal{L}_{legal}(A) \cap \mathcal{L}_{illegal}(M \uparrow \Sigma) = \emptyset$. Informally, this definition says that any legal word w in A can only trigger legal executions in M.
2. **Permissive.** Interface A is permissive for M iff $\mathcal{L}_{legal}(M \uparrow \Sigma) \subseteq \mathcal{L}_{legal}(A)$. Informally, every legal word in M should be represented by some legal word in A.

Note that safety is concerned with blocking behaviors while permissiveness is concerned with including behaviors. These two concepts are complementary in achieving an exact characterization of correct component usage. When dealing with component interfaces, it is therefore important to be able to determine whether a given interface is safe and permissive.

Let $M = \langle Q_M, \alpha M, \delta_M, q_{0M} \rangle$ be the LTS description of a component, and let $A = \langle Q_A, \alpha A, \delta_A, q_{0A} \rangle$ be an interface provided for M.

Checking for safety. Interface A is safe for M if and only if illegal states of M are not reachable in $A \parallel M$. Interface safety can therefore be performed by a reachability check, as supported by any standard model checker. Counterexamples correspond to illegal executions of M that are not blocked by A.

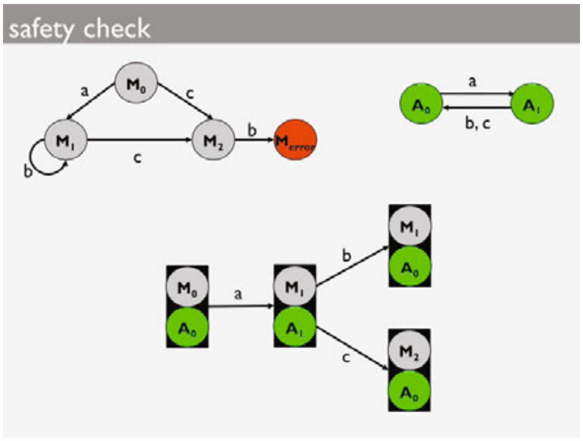


Fig. 2. Safety Check

Figure 2 is used to illustrate the safety check. States M_0, M_1, M_2 and M_{error} belong to component M while states A_0 and A_1 belong to interface A. The error state is no longer reachable in the composition.

Checking for permissiveness. To check permissiveness, we need to complete M with a sink state to obtain M_c, and A with an error state to obtain A_{err}. In $M_c \parallel A_{err}$, we then check for reachability of states that correspond to an error state in A_{err} and a non error state in M_c. A path leading to such states could identify a legal word in M_c that is not accepted by A, reflecting the fact that A is

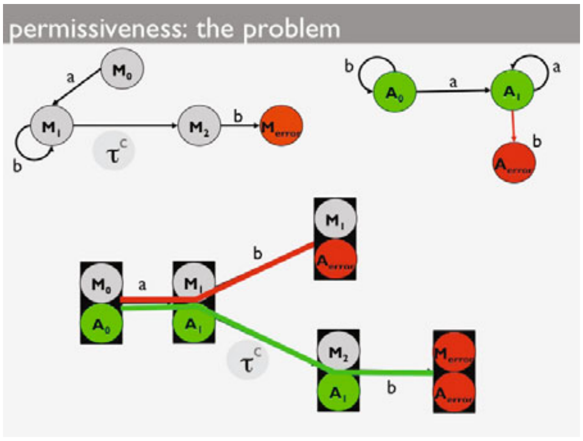


Fig. 3. Checking for Permissiveness

not permissive. However, this check is not sufficient to determine permissiveness of an interface. This is illustrated by an example below.

The example in Figure 3 shows the problem with the permissiveness check above. As before, states M_0, M_1, M_2, M_{error} belong to M and states A_0, A_1 and A_{error} belong to interface A. According to the above check, trace a, b leading to state $[M_1, A_{error}]$ in the composition could be an indication that A is not permissive enough. However this is not true, since the same path leads to $[M_{error}, A_{error}]$. This happens because the alphabet of the assumption is $\{a, b\}$, meaning that action c in M is considered as a τ from the point of view of A. In the figure, this is illustrated as a τ action covering action c.

This example illustrates the fact that non-determinism in component M may cause spurious counterexamples in the permissiveness reachability check described above. As a consequence, precise characterization of permissiveness requires determinization of component M, which can be performed using subset construction. The permissiveness check is therefore NP-hard [1], and can be inefficient in practice.

Several approaches have been proposed to deal with this problem. Unless determinization is a viable solution for a targeted component M[14,3], heuristic approaches are often used to determine whether a counterexample is spurious [1,13]. Also, if non determinism is introduced through abstraction of a deterministic concrete component, this problem can sometimes be avoided, using a combination of over- and under- approximating abstractions [26].

In the next sections we discuss some of these solutions. We first present an approach that creates a safe and permissive interface by construction, and which involves determinization of the component. Subsequently, we describe an iterative learning-based approach that is based on safety and permissiveness checks and which uses heuristics to avoid determinization of the component. We then discuss interface generation in the context of infinite-state components and abstraction.

4 Automated Interface Generation

Precise characterization of component interfaces is a difficult task to perform manually. Given the need for automated modular or compositional verification techniques warranted by the size of modern software and hardware systems, automated interface and assumption generation have been thoroughly investigated in the last decade. Our first attempt to automated interface generation consists of a construction that systematically builds finite-state machine interfaces for finite-state components and safety properties expressed as LTSs [14]. The built interfaces are safe and permissive by construction. Learning-based approaches to interface generation are subsequently discussed. These frameworks are based on the use of the L* algorithm for providing and gradually refining guesses of the desired interface.

4.1 Computing the Weakest Assumption

We describe here an approach to building the weakest assumption for a component with respect to a safety property. The approach addresses the more general problem of model checking for *open* systems, i.e. components that interact with their environments. When model checking a component against a property, our algorithm returns one of the following three results: (i) the component satisfies the property for any environment; (ii) the component violates the property for any environment; or finally, (iii) the "weakest assumption" – an automatically generated assumption that characterizes exactly those environments in which the component satisfies the property.

The traditional approach to verifying a property of an open system is to check it for all the possible environments. The result of verification is either **true**, if the property holds for *all* the possible environments, or **false**, if there exists *some* environment that can lead the component to falsify the property. However this approach may be overly pessimistic and we advocate an optimistic view, which assumes a *helpful* environment. The reason is that software components are often required to satisfy properties only in specific environments, so it is natural to

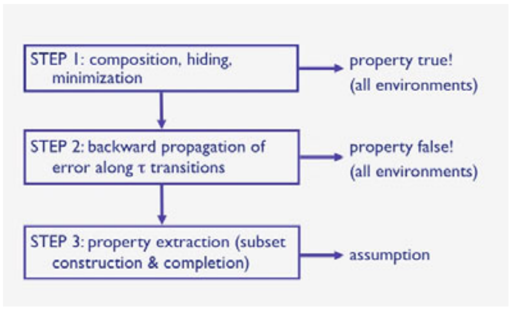

STEP 1: composition, hiding, minimization
property true! (all environments)
STEP 2: backward propagation of error along τ transitions
property false! (all environments)
STEP 3: property extraction (subset construction & completion)
assumption

Fig. 4. Model Checking with Assumption Generation

accept a component if there are *some* environments in which the component does not violate the property.

In our approach, the result of component verification is **true**, if the property holds for *all* environments, similar to the traditional approach. However, the result is **false** only if the property is falsified in *all* environments. If there exist *some* environments in which the component satisfies the property, the result of verification is not false, as in the traditional approach, but rather **true** in environments that satisfy the *weakest assumption*.

Figure 4 illustrates our approach together with the steps we follow to build the weakest assumption (that are described below).

Step 1: Composition and Minimization

Given an *open* system (described as an LTS) and a *property* LTS that may relate the behavior of the system with the behavior of the environment, the first step is to build the *composition* of the system with the *error* LTS of the property and to hide (i.e., turn into τ) the internal actions of the system. The resulting LTS can be minimized with respect to any equivalence that preserves (error) traces.

As an example, let us consider the communication channel from Section 2. We show here the computation of the weakest assumption for component `Input` with respect to property `Order`. Figure 5 depicts the result of composing `Input` with the error automaton for the property. The internal actions of the system, i.e. the transitions labeled `in`, were abstracted to τ. This is illustrated in Figure 5 by covering action `in` with action τ.

If the error state is not reachable in this composition, the property is **true** in any environment, and this is reported to the user. Otherwise, we determine whether there exist environments that can help the system avoid the error; this is achieved through the following steps.

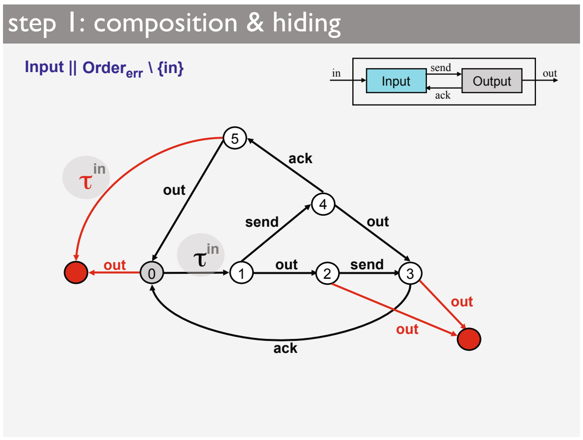


Fig. 5. Step 1: Composition and Hiding

Step 2: Backward Error Propagation

This step first performs *backward propagation* of the error state over τ transitions, thus pruning the states where the environment cannot prevent the error state from being entered via one or more τ steps. We are interested only in the error traces, and therefore we also eliminate the states that are not backward reachable from the error state. If, as a result of this transformation, the initial state becomes an error state, it means that no environment can prevent the system from reaching the error state, so the property is **false** (for all environments) and this is reported to the user.

Consider again the composite system in Figure 5. As a result of backward propagation, we identify state 5 with the error state; the result is shown in Figure 6. The intuition here is that, if the component is in a state from which it can violate the property by some number of internal moves, then no environment can prevent the violation from occurring.

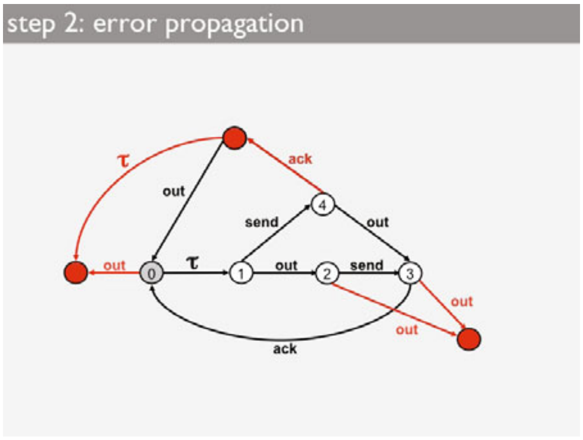


Fig. 6. Step 2: Error Propagation

Step 3: Property Extraction

This step builds the *property* LTS that is our assumption. It performs this in two stages; first it builds the *error* LTS for the assumption, from which it extracts the corresponding property LTS. Note that the LTS resulting from Step 2 might not be an error LTS (i.e. it might not be deterministic or complete), although it contains an error state. Recall from the background section that the error LTS is both deterministic and complete.

In order to get an error LTS we make the LTS obtained from step 2 deterministic by applying to it τ elimination and the subset construction [18], but by taking special care of the π state as follows. During subset construction, the states of the deterministic LTS that is being generated are *sets of states* in the original non-deterministic LTS. If any of these sets contains π, the entire set becomes π. Intuitively, a trace that non-deterministically may or may not lead

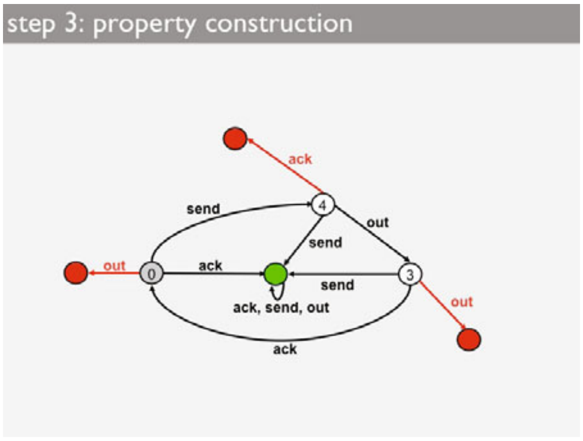


Fig. 7. Step 3: Property Extraction

to an error has to be considered as an error trace. Such non-determinism reflects the fact that, by performing a particular sequence of actions, the environment cannot guarantee that the component will avoid error states.

The resulting LTS is then completed. *Completion* is performed by adding a new "sink" state to the LTS, and adding a transition to this state for each missing transition in the "incomplete" LTS. The missing transitions in the incomplete LTS represent behavior of the environment that is never exercised by the open system under analysis. As a result, no assumptions need to be made about these behaviors. The sink state reflects exactly this fact, since it poses no implementation restrictions to the environment.

The result of subset construction and completion for our running example is shown in Figure 8. The sink state is colored green.

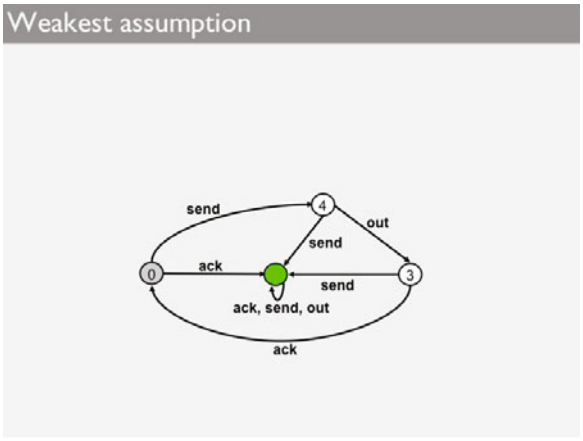


Fig. 8. Computed Assumption

Once we have the error LTS, we obtain the assumption by deleting the error state and the transitions that lead to it.

The assumption for the running example is depicted in Figure 8. The assumption expresses the fact that actions `send`, `out`, `ack` should happen in this order (and this is in fact the encoding of component `Output`); in addition, the assumption allows extra behaviors (the ones that lead to the sink state). It can be shown that indeed this assumption is the "weakest".

4.2 Learning Component Interfaces

Let $M = \langle Q_M, \alpha M, \delta_M, q_{0M} \rangle$ be a component, and $\Sigma \subseteq \alpha M$ denote the communication alphabet of component M, i.e., the set of actions through which M communicates with its environment. Our goal is to compute M's precise interface as a finite state automaton A over Σ, in other words an interface A that is both *safe* and *permissive*, as defined in Section 3.

Since A represents a regular language, we can use the learning algorithm L* to learn it. To this aim, we need to provide L* with a teacher that represents the language of A. As discussed in the following, the teacher can be implemented using model checking, since all questions asked by L* can be reduced to reachability problems (see Figure 9).

Queries. L* is first used to repeatedly *query* M to check whether, in the context of strings s, M reaches an error state. If it does, then s corresponds to an illegal execution of M and should be excluded from A and the query returns *false*. Otherwise, s should be included in A, and the query returns *true*. If error states are introduced by some property P, then the query corresponds to checking the triple $\langle s \rangle\ M\ \langle P \rangle$ as illustrated in Figure 9 (we abuse notation here, and use s to represent $lts(s)$).

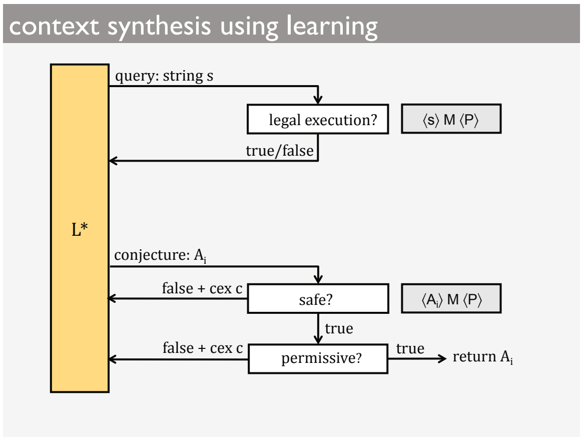


Fig. 9. Learning for Context Synthesis

Conjectures. The conjectured automaton A is checked for correctness, which in this context means checking whether the interface that it represents is safe and permissive. We therefore break down answering conjectures into two parts:

Oracle 1. Checks if A is *safe*, using the model checking procedure described in Section 3. Again, if error states are introduced by some property P, the safety checks corresponds to checking the triple $\langle A \rangle\ M\ \langle P \rangle$, as illustrated in Figure 9. If A is safe, then the teacher proceeds to Oracle 2. If it is unsafe, the model checker returns a counterexample t. The resulting counterexample t, projected on the interface alphabet Σ, is returned to L* to refine its conjecture (see Figure 9, where $c = t \uparrow \Sigma$). The projection is necessary because L* needs counterexamples in terms of the alphabet over which it is learning.

Oracle 2. Checks if safe interface A is also permissive, using the model checking procedure described in Section 3. If the interface is permissive, then the framework terminates with A as a safe, permissive and minimal interface for M (minimality is guaranteed by the characteristics of the L* algorithm). If, on the other hand, a counterexample t is returned, then this may be because the interface needs to be refined, or it could be because the permissiveness procedure is not precise in the presence of non-determinism. As discussed above, one could determinize component M for performing this check. Other, more light-weight approaches propose heuristics.

For example, one such heuristic consists of making a *query* on $c = t \uparrow \Sigma$ (with the same mechanism as L* queries are answered). If the query returns true, then it means the interface is not permissive, and therefore c is returned to L* for refinement, and the learning process continues with more queries and eventually with a new conjecture (see Figure 9).

If the query returns false, then c does not correspond to a real counterexample. Model checking therefore ignores this state. Several approaches have been proposed at this point. One approach applies an additional heuristic step [1], whereas another backtracks after the spurious counterexample and continues the state space exploration [13]. The latter approach is illustrated in Figure 10. This heuristic is non-trivial to implement within a model checker. The reason is that the permissiveness check consists of a reachability check within which a query is invoked to potentially invalidate a discovered counterexample, in which case the reachability check backtracks and continues the search. In essence, querying within a reachability check would naively mean that a model checker is invoked within a model checker, which is clearly inefficient.

For this reason, all query results are stored in a memoized table which is consulted during reachability analysis. If a potential counterexample discovered is stored in the memoized table as a spurious one, then the algorithm backtracks and tries a different path. If it is a real counterexample, it is returned to L*. If it is not stored in the table, the reachability check terminates. A query then follows as an independent step, and the reachability check is started from scratch. Since the result of the query is stored in the memoized table, the reachability check is guaranteed to return a different counterexample in the next round.

```
permissiveness check
MC: model check for ($M_i$,$A_{error}$)
    reached ($M_i$,$A_{error}$) by trace t
    c = t $\uparrow$ $\Sigma$
    if (memoized(c) == false) // c is spurious
        backtrack and continue search
    else  // memoized(c) == true or c not in memoized
        model checker produces c
if (query(c) == yes)
    return c to L* // not permissive
else restart at MC
```

Fig. 10. Permissiveness heuristic

Non-determinism: In summary, unless component M is deterministic with respect to the alphabet Σ of the assumption, precise interfaces can only be computed through determinization of M, which may result in a component exponentially larger than M. Heuristics can be quite efficient at getting a precise interface, but cannot provide guarantees. Of course, if the permissiveness check does not encounter spurious counterexamples, then we know that the resulting interface is precise, despite heuristics.

There are two cases where determinization can be avoided, as will be described in the following sections. First, when the potential source for non-determinism is abstraction of a deterministic infinite state component, then we may use a combination of over- and under- approximations to precisely compute a component interface while avoiding determinization in the permissiveness step. Second, when the environment of a component is available, determinization can be avoided during compositional reasoning. Since in this context, rather than a precise interface for the component irrespective of environment, we just need the interface to act as an assumption for an assume-guarantee rule, the environment can be used to selectively increase the interface permissiveness. We discuss these cases in the following sections.

5 Interface Generation and Abstraction

The learning frameworks that we discussed in the previous section only apply to finite state components since they rely on teachers that exhaustively explore the component state space. However, most realistic components are infinite state for all practical purposes. A typical approach for dealing with large components in model checking is by using abstraction techniques. In this section, we discuss a framework that computes interfaces of potentially infinite-state components by combining abstraction and learning approaches.

There are two types of abstractions that one may built of a component. An over-approximation ("may" abstraction) is an abstraction that contains a superset of the behaviors of the component. The advantage of over-approximations is that, when used for checking properties, if the property is satisfied for the overapproximation, then it is also satisfied for the concrete component. The disadvantage is that when a counterexample is obtained, it may be a spurious one, since it may correspond to a behavior that is not really feasible in the concrete component. These characteristics are reversed for under-approximations. An under-approximation ("must" abstraction) contains only a subset of the behaviors of the concrete component. As such, it will only return real counterexamples. In the absence of errors, however, there is not guarantee that the concrete component is also error-free since there may be concrete behaviors that are not accounted for in the under-approximation.

The may and must abstractions that we use [26] are obtained using predicate abstraction. Predicate abstraction is a technique that substitutes component variables on large or infinite domains with a finite set of predicates over these variables. Concrete states of the component are then substituted with abstract states representing valuations of the selected predicates. In a may abstraction, an abstract transition links two abstract states if there exists a concrete transition between concrete states represented by the two abstract states. May transitions between abstract states may or may not correspond to actual transitions in the concrete system. A may abstraction is an over-approximation (see Figure 11). On the other hand, in a must abstraction, abstract transitions link two abstract states only if all the concrete states represented by the two abstract states are linked by concrete transitions. Must transitions are guaranteed to represent transitions in the concrete system, but do not necessarily cover all concrete transitions. A must abstraction is therefore an under-approximation of the concrete component (see Figure 11).

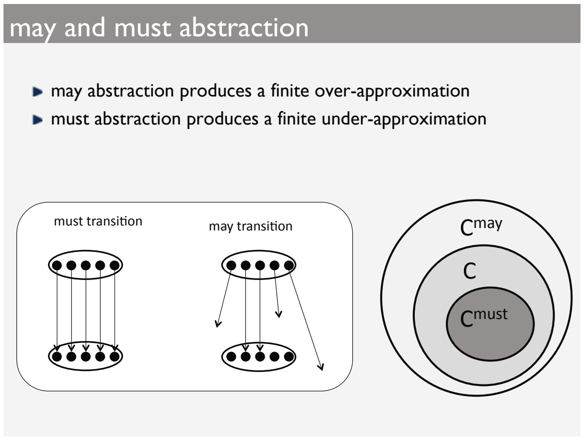

may and must abstraction
▸ may abstraction produces a finite over-approximation
▸ must abstraction produces a finite under-approximation
must transition
may transition
C^{may}
C
C^{must}

Fig. 11. May and Must Abstraction

Let C be a component corresponding to a potentially infinite-state transition system S_C. From now on, for simplicity, we will use C to represent the component and its transition system. We have proposed interface-generation algorithms that operate by analyzing *finite-state* abstractions of C [26]. The essence of our approach lies in the following observation:

Theorem 1. *Assume a component C, a may abstraction $C^{may}$and a must abstraction $C^{must}$for C. If an interface $A$for C is permissive with respect to C^{must} and safe with respect to C^{may}, then $A$is safe and permissive with respect to C.*

To provide an intuition for this theorem, let us analyze the relationships between the languages corresponding to may and must abstractions. For any component C, since C^{may}has more behaviors that C, it follows that $\mathcal{L}_{illegal}(C) \subseteq \mathcal{L}_{illegal}(C^{may})$, and consequently, $\mathcal{L}_{legal}(C) \supseteq \mathcal{L}_{legal}(C^{may})$. On the other hand, $\mathcal{L}_{illegal}(C) \supseteq \mathcal{L}_{illegal}(C^{must})$, and consequently, $\mathcal{L}_{legal}(C) \subseteq \mathcal{L}_{legal}(C^{must})$. If an interface A is safe with respect to C^{may}, it means that its legal executions are a subset of the legal executions of C^{may}. Similarly, if A is permissive, its illegal executions are a subset of the illegal executions of C^{must}. These relationships are illustrated in Figure 12. Given the complementary nature of the legal and illegal execution sets of any component, an interface can only have both properties if $\mathcal{L}_{illegal}(C^{must}) = \mathcal{L}_{illegal}(C) = \mathcal{L}_{illegal}(C^{may}) = \mathcal{L}_{illegal}(A)$.

Our approach for interface generation is therefore based on constructing may and must abstractions for a concrete component $C(C^{may}$and C^{must}, respectively). Its novelty with respect to previous work is that it uses $C^{may}$to check whether an interface is safe, and $C^{must}$to check whether an interface is permissive. The advantage of the approach is that, if the concrete component is deterministic, then so is C^{must}, since it under-approximates the concrete behavior. By using $C^{must}$for the permissiveness check, we therefore avoid determinizing

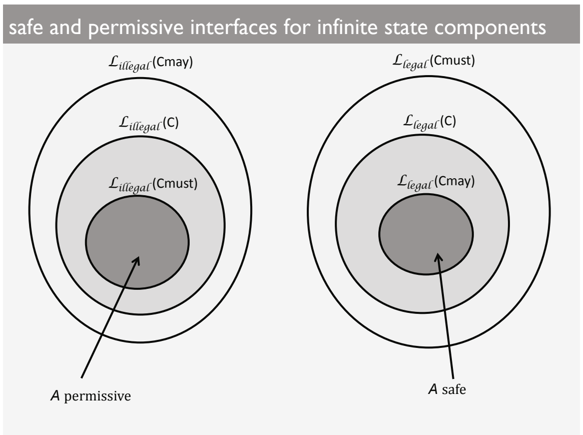

safe and permissive interfaces for infinite state components
$\mathcal{L}_{illegal}$(Cmay)
$\mathcal{L}_{illegal}$(C)
$\mathcal{L}_{illegal}$(Cmust)
A permissive
$\mathcal{L}_{legal}$(Cmust)
$\mathcal{L}_{legal}$(C)
$\mathcal{L}_{legal}$(Cmay)
A safe

Fig. 12. Relationship between languages of abstractions of a concrete component and the component itself

the abstractions that are constructed, while still providing guarantees for safety and permissiveness of the computed interface.

5.1 Learning Interfaces Using Abstractions

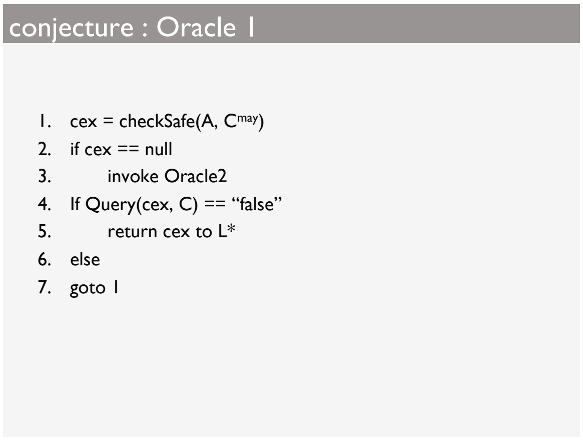
In order to compute an interface for component C, we use the learning framework presented in the previous section. The teacher is very similar, except that it sometimes needs to trigger a refinement of the abstraction, in order to provide answers to the learner. Refinement consists of adding predicates, and it is performed on demand, within the teacher's mechanism for answering the L* questions. More specifically, the teacher operates as follows.

Queries. The procedure for queries is illustrated in Figure 13. It first checks whether the word σ triggers an illegal execution in C^{must}. If it does, σ should not belong to A because it must also trigger an illegal execution in C. So the query returns *false*. Otherwise, σ is checked against C^{may}. If it is safe for C^{may}, then σ must belong to A so the query returns *true*. Otherwise, we have a situation where σ is safe for $C^{must}$and unsafe for C^{may}. In other words, σ demonstrates that the illegal languages of $C^{may}$and $C^{must}$are not equal. As discussed earlier in the section, we are able to compute an interface when the illegal languages of $C^{may}$and C^{must}become equal. We therefore need to refine the abstraction, and check the query again. L* is not involved in the refinement or restarted after it; it just awaits for the teacher to come up with a response to the query. The response is always consistent with the concrete component C.

Conjectures. We use Theorem 1 to answer the conjectures using two oracles, as illustrated in Figure 14 and Figure 15.

```
Query(σ, C)

if checkSafe(σ,C^must) != null
      return "false"
cex = checkSafe(σ,C^may)
if cex == null
      return "true"
Preds = Preds U Refine(cex)
Query(σ, C)
```

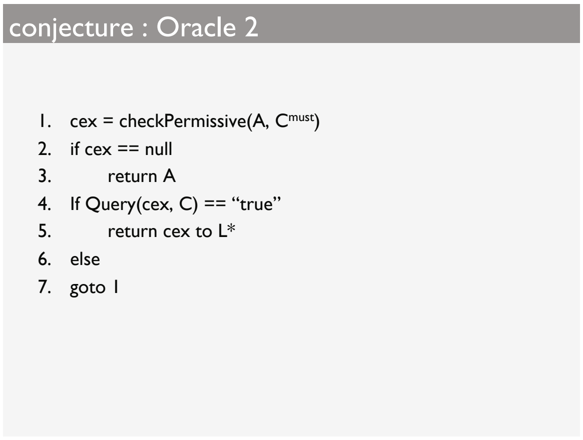
Fig. 13. Answering queries



conjecture : Oracle 1

```
1. cex = checkSafe(A, $C^{may}$)
2. if cex == null
3.       invoke Oracle2
4. If Query(cex, C) == “false”
5.       return cex to L*
6. else
7. goto 1
```

Fig. 14. Answering conjectures: Oracle 1



conjecture : Oracle 2

```
1. cex = checkPermissive(A, $C^{must}$)
2. if cex == null
3.       return A
4. If Query(cex, C) == “true”
5.       return cex to L*
6. else
7. goto 1
```

Fig. 15. Answering Conjectures: Oracle 2

Oracle 1 is invoked first. If it finds that A is safe with respect to C^{may}, Oracle 2 gets invoked. If Oracle 2 finds that A is also permissive with respect to C^{must}, we conclude from Theorem 1 that A is a safe and permissive interface for C. All remaining cases require either the refinement of A by L*, or the refinement of the component abstractions. We use queries to help us determine what needs to be refined. Our approach is described in detail below.

Oracle 1: If A is not safe with respect to C^{may}, we obtain a counterexample *cex*, which is allowed by A but leads to error in C^{may}. We subsequently query *cex* in order to determine whether it is indeed a counterexample to the safety of A. Note that the querying procedure may involve refinement of the abstraction.

If the query returns no, then it means that *cex* should indeed not be in the language of A, so *cex* is returned to L* for A to be refined. Otherwise, we invoke Oracle 1 again, knowing that Predshave been updated because abstraction refinement must have occurred.

Oracle 2: If A is not permissive with respect to C^{must}, we obtain a counterexample *cex*, which corresponds to a word that is not allowed by A. We subsequently query *cex* in order to determine whether it is indeed a counterexample to the permissiveness of A. If the query response is positive, then *cex* should belong to A, so *cex* is returned to L* for refining the assumption. Note again that querying may involve refinement. If the response in negative, then the permissiveness check is invoked again, because we know there must have been abstraction refinement involved.

More details and explanations are provided in [26].

5.2 Applicability and Related Approaches

The learning scheme presented in this section for computing interfaces of infinite state components generates deterministic finite state automata. As such, its applicability is restricted to interfaces that can be represented in this fashion. The framework that we have developed may not always termine, which is always a possibility in abstraction refinement schemes. However, if the concrete component C has a finite bisimulation quotient, then our framework is guaranteed to terminate and produce a minimal safe and permissive interface for C [26].

Other related approaches to inteface generation for infinite components have been presented in [1,17]. Both approaches construct only over-approximations of the component behavior, which may be non-deterministic. As mentioned, checking permissiveness when (abstracted) components are non-deterministic requires a potentially expensive determinization step. Alur et al. [1] avoid this step by using heuristics, and therefore cannot guarantee permissiveness of the generated interfaces. On the other hand, Henzinger et al. [17] build “abstract regions”, which is equivalent to performing a determinization step. Furthermore, the abstraction mechanisms in [17] cannot guarantee minimal interfaces. Even if these interfaces were to be minimized, this approach would suffer from potentially large intermediate interfaces that subsequently get compacted. This latter problem is more pronounced in the presence of the determinization step, which is exponential, in the worst case. In contrast, L*-based approaches like ours and [1] directly generate minimal interfaces. Note however that the technique by [1] does not provide criteria to automatically detect the need for abstraction refinement. Their refinements are based on inspection of the generated interfaces, and are performed manually. In contrast, refinement in our work [26] is performed automatically.

6 Assumption Generation for Compositional Verification

As discussed in Section 2, assume guarantee reasoning provides solutions to the problem of decomposing the verification of a large system into local verification steps of the system components. The most challenging part of applying assume-guarantee reasoning, however, is coming up with appropriate assumptions to use in the application of the assume-guarantee rules. In this section, we discuss work on generating assumptions for automated assume-guarantee verification.

We will restrict ourselves to the simple rule presented in Section 2, and will then discuss how one can expand to other rules.

As discussed earlier in this paper, the weakest assumption captures precisely all restrictions that a component needs to make on its environment in order to satisfy some safety property(ies). The weakest assumption can safely be used for assume-guarantee reasoning; in fact, with the weakest assumption, the rule also becomes complete since, the second premise holds ($\langle true \rangle\ M_2\ \langle A \rangle$) if and only if the conclusion of the rule holds ($\langle true \rangle\ M_1 \parallel M_2\ \langle P \rangle$).

The weakest assumption corresponds to a safe and permissive interface for component M_1. We could therefore use L* to learn this assumption while automatically verifying a property on some system in an assume guarantee style. The framework of Figure 16 demonstrates the steps involved in performing automated assume-guarantee reasoning while learning the weakest assumption. Queries are asnwered in the exact same fashion as in the interface generation framework of Figure 9. The first Oracle, checking whether the conjecture corresponds to a safe interface for M_1, is answered identically to the interface generation framework. Note that checking for safety corresponds to checking the first premise of the assume-guarantee Rule ASym.

In terms of the permissiveness check, we can now take advantage of the fact that M_2 is available, to avoid determinization of M_1. Remember that our main target in this framework is to prove or disprove a property on the system using assume-guarantee reasoning. Since Oracle 1 checks that premise 1 of Rule ASym holds, it remains to check premise 2 ($\langle true \rangle\ M_2\ \langle A \rangle$). Premise 2 therefore substitures Oracle 2 of the original interface generation framework. If this check

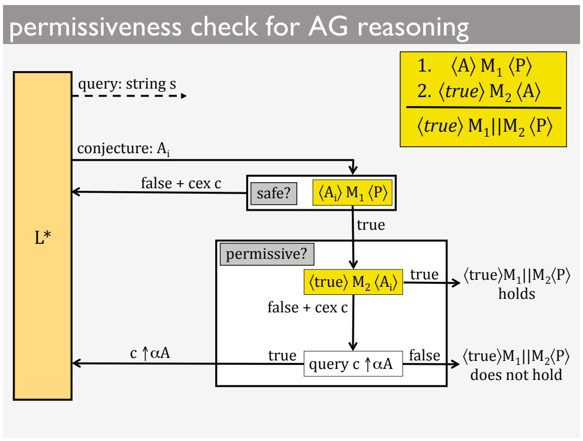


Fig. 16. Learning Assumptions for Assume-Guarantee Reasoning

passes, then we know that both premises of Rule ASym hold, and therefore the property holds for $M_1 \parallel M_2$.

If the check fails, the Teacher performs some analysis to determine the underlying reason (see Figure 16). The Teacher performs a query (of the L* type) in order to determine whether the returned counterexample *cex*, projected to the alphabet of the assumption, should belong to the conjectured assumption A. If the answer is true, meaning that $c \uparrow \alpha A$ should be included in A, then it means that A is not the weakest assumption since it does not include a safe word, and $c \uparrow \alpha A$ is returned to L* for refinement of A. If, on the other hand, the answer is false, it means that c is a word that belongs to M_2, in the context of which M_1 violates the property P. As a consequence, $M_1 \parallel M_2$ does not satisfy the property P.

Notice that the answers that this modified Teacher provides to L* are always with respect to the weakest assumption. However, the framework uses M_2 to filter which missing words to include in the language of the assumption, as opposed to adding all of them. The reason is that we restrict our reasoning to a specific context, rather than accounting for all possible contexts. As a result, we no longer require determinization of component M_1.

Note also that we do not always obtain the weakest assumption from this framework; in other words, the obtained assumption is not the most permissive. Our primary goal is to obtain conclusive results from the assume guarantee rule. As soon as we are able to prove or disprove the property in the system, we stop refining the learned assumptions. At that point, we may, or may not have reached the weakest assumption. We will however have reached an assumption that completes our verification; this assumption is smaller than or equal to the weakest assumption, as guaranteed by the characteristics of L*. For our running example, the assumption generated is smaller than the weakest assumption, as illustrated in Figure 17. The second conjecture, A_2, generated by L*, passes

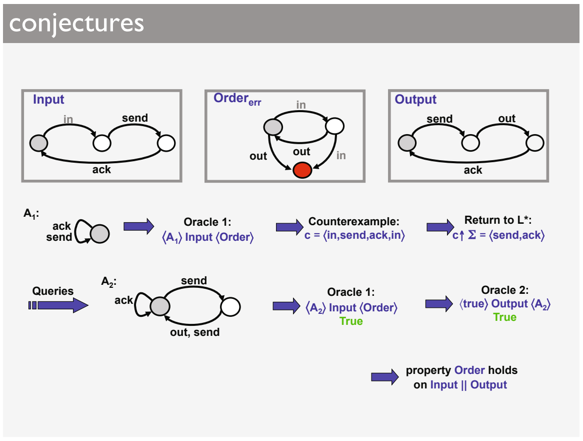


Fig. 17. Assumption for AG reasoning

both Oracles and the learning framework terminates reporting that the property holds; notice that A_2 has only two states as compared to the weakest assumption that has four states (see Figure 8).

Given the fact that our Teacher only comes back to L* for refinement with counterexamples for the weakest assumption, the framework will eventually converge to the weakest assumption unless it terminates earlier. We have shown [23] that with the weakest assumption, the rule becomes sound and complete, and therefore our framework will return a conclusive answer at that iteration. As a result, the framework always terminates.

To summarize, we presented a framework that computes an assumption for automated assume-guarantee reasoning. We cannot tell if the framework computes the weakest assumption, but we know that it will do so if necessary, and thus guarantees termination.

6.1 Related Approaches

We have showed how to guide our learning of assumptions for compositional verification towards the weakest assumption. Other researchers focused on the more computationally expensive problem of learning a *minimal* assumption [16,8] for compositional verification. In other words, computing an assumption A_{min} such that any other assumption A that can check satisfaction or violation of P will have a greater than or equal number of states, i.e., $|A| \geq |A_{min}|$.

The alphabet of the assumptions we learn for compositional verification is fixed to $(\alpha M_1 \cup \alpha P) \cap \alpha M_2$. Other researchers and ourselves observed that it may sometimes be possible to verify a problem with a smaller alphabet, and therefore potentially smaller assumptions [23,6]. Learning assumptions can be extended for other assume-guarantee rules that are symmetric and may involve circularity, and may involve multiple components [23]. Rule ASym itself can be extended to multiple components through recursive invocation. Learning has also been applied in the context of symbolic and implicit model checking [21,7], and of assume-guarantee reasoning for liveness properties [11].

7 Discussion and Conclusions

In this tutorial paper, we reviewed several approaches for context synthesis. Our context synthesis techniques rely on standard model checking features, such as reachability analysis and counterexample generation. Over the years, we have implemented our techniques on top of several well known model-checkers, such as LTSA [14], SPIN [22], Java PathFinder [13], and ARMC [26]. We have experimented with compositional verification and interface generation techniques in the context of several applications, mostly involving NASA systems. Our NASA case studies include a Rover Executive [10,15,4], autonomous rendez vous and docking [5], a resource arbiter for the Mars Exploration Rover [12], and models of the flight phases of a spacecraft [13,26]. We have also experimented with existing benchmarks for compositional verification [12] and for interface generation [26].

In our experience, learning-based interface generation and compositional verification were most successful when a system has a well-designed component-based structure, where component interfaces are small. Beyer, Henzinger and Singh make a similar observation [3]. Moreover, even though abstraction can be introduced in order to deal with large component implementations, it is still much harder to generate interfaces at the level of source code. Interface are ideally generated at design time, and are then used in several ways throughout the life cycle of a component: for compositional verification, concrete component and system integration testing, runtime verification, and incremental verification in the presence of component upgrades or substitutions.

Our learning based algorithms for context synthesis are part of the open-source Java PathFinder tool-set and they are available from the following website: `http://babelfish.arc.nasa.gov/trac/jpf/`, the `jpf-cv` project.

Our work on interface generation needs to be extended and matured in order to make it applicable in practice. There are several interesting future research directions. Some of them involve the generation of interfaces that go beyond purely functional properties such as safety and liveness, but potentially timed or probabilistic properties. Moreover, it would be interesting to try and identify design decisions that facilitate the generation of component interfaces. Finally, one could investigate interfaces in different domains such as service-oriented systems, aerospace systems, and others.

References

1. Alur, R., Cerný, P., Madhusudan, P., Nam, W.: Synthesis of interface specifications for Java classes. In: Palsberg, J., Abadi, M. (eds.) POPL, pp. 98–109. ACM, New York (2005)
2. Angluin, D.: Learning regular sets from queries and counterexamples. Inf. Comput. 75(2), 87–106 (1987)
3. Beyer, D., Henzinger, T.A., Singh, V.: Algorithms for interface synthesis. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 4–19. Springer, Heidelberg (2007)
4. Blundell, C., Giannakopoulou, D., Pasareanu, C.S.: Assume-guarantee testing. ACM SIGSOFT Software Engineering Notes 31(2) (2006)
5. Brat, G., Denney, E., Giannakopoulou, D., Jonsson, A.: Verification of autonomous systems for space applications. In: IEEE Aerospace Conference (2006)
6. Chaki, S., Strichman, O.: Three optimizations for assume-guarantee reasoning with L*. Formal Methods in System Design 32(3), 267–284 (2008)
7. Chen, Y.-F., Clarke, E.M., Farzan, A., Tsai, M.-H., Tsay, Y.-K., Wang, B.-Y.: Automated assume-guarantee reasoning through implicit learning. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 511–526. Springer, Heidelberg (2010)
8. Chen, Y.-F., Farzan, A., Clarke, E.M., Tsay, Y.-K., Wang, B.-Y.: Learning minimal separating DFAs for compositional verification. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 31–45. Springer, Heidelberg (2009)
9. Clarke, E., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press, Cambridge (1999)

10. Cobleigh, J.M., Giannakopoulou, D., Pasareanu, C.S.: Learning assumptions for compositional verification. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 331–346. Springer, Heidelberg (2003)
11. Farzan, A., Chen, Y.-F., Clarke, E.M., Tsay, Y.-K., Wang, B.-Y.: Extending automated compositional verification to the full class of omega-regular languages. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 2–17. Springer, Heidelberg (2008)
12. Gheorghiu, M., Giannakopoulou, D., Pasareanu, C.S.: Refining interface alphabets for compositional verification. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 292–307. Springer, Heidelberg (2007)
13. Giannakopoulou, D., Pasareanu, C.S.: Interface generation and compositional verification in javaPathfinder. In: Chechik, M., Wirsing, M. (eds.) FASE 2009. LNCS, vol. 5503, pp. 94–108. Springer, Heidelberg (2009)
14. Giannakopoulou, D., Pasareanu, C.S., Barringer, H.: Component verification with automatically generated assumptions. Autom. Softw. Eng. 12(3), 297–320 (2005)
15. Giannakopoulou, D., Pasareanu, C.S., Cobleigh, J.M.: Assume-guarantee verification of source code with design-level assumptions. In: ICSE, pp. 211–220 (2004)
16. Gupta, A., McMillan, K.L., Fu, Z.: Automated assumption generation for compositional verification. Formal Methods in System Design 32(3), 285–301 (2008)
17. Henzinger, T.A., Jhala, R., Majumdar, R.: Permissive interfaces. SIGSOFT Softw. Eng. Notes 30, 31–40 (2005)
18. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation. Adison-Wesley Publishing Company, Reading (1979)
19. Jones, C.B.: Tentative steps toward a development method for interfering programs. ACM Trans. Program. Lang. Syst. 5(4), 596–619 (1983)
20. Magee, J., Kramer, J.: Concurrency: state models & Java programs. John Wiley & Sons, Inc., New York (1999)
21. Nam, W., Madhusudan, P., Alur, R.: Automatic symbolic compositional verification by learning assumptions. Formal Methods in System Design 32(3), 207–234 (2008)
22. Pasareanu, C.S., Giannakopoulou, D.: Towards a compositional SPIN. In: Valmari, A. (ed.) SPIN 2006. LNCS, vol. 3925, pp. 234–251. Springer, Heidelberg (2006)
23. Pasareanu, C.S., Giannakopoulou, D., Bobaru, M.G., Cobleigh, J.M., Barringer, H.: Learning to divide and conquer: applying the L* algorithm to automate assume-guarantee reasoning. Formal Methods in System Design 32(3), 175–205 (2008)
24. Pnueli, A.: In transition from global to modular temporal reasoning about programs, pp. 123–144. Springer-Verlag New York, Inc., New York (1985)
25. Rivest, R.L., Schapire, R.E.: Inference of finite automata using homing sequences. Inf. Comput. 103(2), 299–347 (1993)
26. Singh, R., Giannakopoulou, D., Pasareanu, C.S.: Learning component interfaces with may and must abstractions. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 527–542. Springer, Heidelberg (2010)

Middleware-Layer Connector Synthesis: Beyond State of the Art in Middleware Interoperability

Valérie Issarny[1], Amel Bennaceur[1], and Yérom-David Bromberg[2]

[1] INRIA, CRI Paris-Rocquencourt, France
[2] LaBRI, University of Bordeaux, France

Abstract. This chapter deals with interoperability among pervasive networked systems, in particular accounting for the heterogeneity of protocols from the application down to the middleware layer, which is mandatory for today's and even more for tomorrow's open and highly heterogeneous networks. The chapter then surveys existing approaches to middleware interoperability, further providing a formal specification so as to allow for rigorous characterization and assessment. In general, existing approaches fail to address interoperability required by today's ubiquitous and heterogeneous networking environments where interaction protocols run by networked systems need to be mediated at both application and middleware layers. To meet such a goal, this chapter introduces the approach that is investigated within the CONNECT project and that deals with the dynamic synthesis of *emergent connectors* that mediate the interaction protocols executed by the networked systems.

Keywords: Interoperability, Middleware, Pervasive networking, Protocol mediation.

1 Introduction

As networked systems are becoming increasingly pervasive, they need to compose dynamically with their ever evolving environment according to functionalities they provide and/or request. However, such dynamic composition is greatly challenged by the heterogeneity and autonomy of today's digital systems, which are not designed in concert, but are instead independently developed and deployed within pervasive networking environments. As a result, although networked systems may possibly match from the standpoint of provided and required functionalities, actual behavioral matching is unlikely due to inherent design diversity. Therefore, what is needed for enabling the composition of pervasive networked systems is *emergent connectors* [28], which embed a mediation process so as to adapt the systems' respective interaction behaviors for the sake of coordination.

The notion of *mediator* underlying emergent connectors is not new. It has indeed been investigated since the need for interoperability in distributed systems was identified [23]. However, this was initially a design-time concern, while today's dynamic distributed systems require on-the-fly mediation. On-the-fly protocol mediation has in particular been studied quite extensively in the context of Web services to deal with either dynamic service composition (e.g., [14]) or substitution (e.g., [12]). Still, as in particular investigated in the companion chapter on application-layer connector synthesis [26], existing work on runtime automated mediation concentrates on application-layer protocols, while the heterogeneity of open networked systems may concern both the application and middleware layers.

As surveyed within companion chapter on interoperability in complex distributed systems [6], middleware interoperability solutions have been developed since the early days of middleware. While one-to-one bridging was among the early approaches [40], it evolved into more generic solutions such as Enterprise Service Bus [13], interoperability platforms [21] and transparent interoperability approaches [9,36]. However, except for the transparent interoperability approaches, most of these solutions rely upon the design-time choice to develop applications using the proposed interoperability solution. Thus, they do not allow for on-the-fly interoperability between networked applications embedding different legacy middleware. Middleware interoperability further needs to cope with the many middleware interaction paradigms that now need to coexist. This includes accessing the same functionality through distinct paradigms (e.g., context-awareness through access to a data-centric sensor network or an RPC-based context server).

As an illustration, consider the simple, yet challenging scenario of photo sharing within a public space such as a stadium, which is also investigated from the standpoint of application-layer connection in [26]. Typically, the target environment allows for both infrastructure-based and ad hoc peer-to-peer photo sharing. In the former implementation, a photo sharing service is provided by the stadium, where only authenticated photographers are able to produce pictures while any spectator may download and even annotate pictures. The peer-to-peer implementation allows for photo download, upload and annotation by any spectator, who are then able to directly share pictures using their handhelds. In both cases, the spectator's handheld would need to embed the appropriate software application, which may not be available due to the handheld's specific platform. Further, the spectator may not be willing to download yet another photo sharing application, i.e., the proprietary implementation offered by the stadium, while one is already available on the handheld. Moreover, while the photo sharing functionality is present in both versions of the photo sharing application, it is unlikely that they feature the very same interface and behavior. In particular, the RPC interaction paradigm suits quite well the infrastructure-based service, while a distributed shared data space is more appropriate for the peer-to-peer version. In general, considering the ever-growing base of content-sharing applications for handhelds, numerous versions of the photo sharing application may be available on the spectators' handhelds, thus calling for appropriate interoperability solutions that mediate interaction protocols from the application down to the middleware layer.

This chapter more specifically concentrates on middleware-layer interoperability, i.e., enabling networked systems that functionally match to be able to coordinate despite running heterogeneous middleware protocols. The next section formalizes the role of middleware in the connection of networked systems, in particular highlighting the inter-play between application- and middleware-layer protocols. Then, Section 3 focuses on *interoperability connectors* introduced in the literature, as surveyed in companion chapter [6]; the behavior of interoperability connectors is formally defined, hence providing a rigorous characterization of their respective features. As presented, interoperability connectors allow overcoming the heterogeneity of middleware protocols as long as the protocols implement the same coordination paradigm, which is too restrictive regarding the objective of enabling emergent connectors. Section 4 paves the way for enabling emergent connectors, i.e., the on-the-fly synthesis of connectors that mediate interaction protocols from the application down to the middleware layer, which builds upon the theory of mediators presented in companion chapter [26]. Finally, Section 5 concludes with perspective for future work towards effecting emergent middleware.

2 Middleware-Based Connectors

In the context of distributed systems, a connector abstracts a complex interaction behavior that is facilitated by middleware which provides services to realize this interaction. In particular, middleware overcomes the heterogeneity of the distributed infrastructure by establishing a software layer that homogenizes the infrastructure's diversities using a well-defined and structured distributed programming model [27]. In particular, middleware induces an interaction paradigm for enabling distributed networked systems to coordinate [38].

In the following, we introduce middleware-based connectors using state-of-the-art connector classification [34,50] (Section 2.1) and define their specification using formal notation(Section 2.2). Then, we describe the mismatches preventing connection among components and introduce the needed mediation to enforce interoperability among them.

2.1 A Classification of Middleware-Based Connectors

Based upon the classification of connectors introduced in [34,50], services provided by middleware are depicted in Figure 1. The *communication* and *coordination* services support the transfer of data and control among components and can be realized by different connector types, each of which defining an *interaction paradigm* such as *procedure call*, *event*, *message-based*, or *data access* connectors. The *adaptor* connector type provides a *conversion* service to support interaction among heterogeneous components while services that *facilitate* interaction among components are achieved using the *distributor* and *arbitrator* connector types. Distributor connectors perform discovery through the identification of interaction paths and subsequent routing of communication and coordination

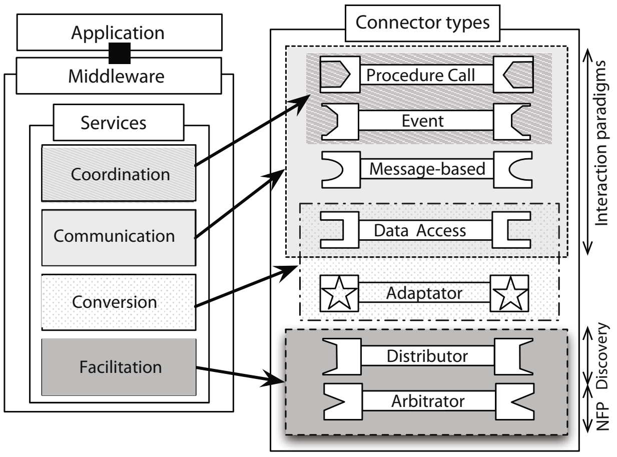


Fig. 1. Middleware-based connector classification

information among components along these paths. Non-functional properties (NFP) are managed by *arbitrator* connectors that streamline system operations, resolve any conflict and redirect the flow of control.

Each connector type is associated with different dimensions (and subdimensions) representing its architectural details. For example, a procedure call connector defines the *Parameters* dimension that is subdivided into *data transfer*, *semantics*, *return value*, and *invocation record* subdimensions. The procedure call connector type is also associated to other dimensions such as *Entry point* associated to two subdimensions, *single* or *multiple*, *Invocation* defining the *implicit* and *explicit* subdimensions, *Synchronicity*, *Cardinality*, and *Accessibility.* The values associated to the various dimensions and subdimensions define a connector implementation, that is, a specific middleware. For example, SOAP[1] (Simple Object Access Protocol), CORBA[2] (Common Object Request Broker Architecture), and RMI[3] (Remote Method Invocation) are specific middleware defining implementations of the procedure call connector type.

2.2 Formalizing Middleware-Based Connectors

In order to precisely characterize the role of middleware in the connection of networked systems, this section formalizes middleware-based connectors using FSP [33], as FSP has proven to be a convenient formalism for specifying connectors [47]. In particular, using FSP allows us to exploit the LTSA tool [33] to automate reasoning about the behavior of connectors and connected systems.

[1] http://www.w3.org/TR/soap/
[2] http://www.omg.org/technology/documents/corba_spec_catalog.htm
[3] http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html

Table 1. FSP syntax overview

Definitions	
END	Predefined process, denotes the state in which a process successfully terminates
set S	Defines a set of action labels
$[i : S]$	Binds the variable i to a value from S
Primitive Processes (P)	
$a \rightarrow P$	Action prefix
$a \rightarrow P \vert b \rightarrow P$	Choice
$P; Q$	Sequential composition
$P(X =' a)$	Parameterized process: P is described using parameter X and modeled for a particular parameter value, $P(a1)$
$P/\{new\_1/old\_1, ..., new\_n/old\_n\}$	Relabeling
$P \backslash \{a_1, a_2, ..., a_n\}$	Hiding
$P + \{a_1, a_2, ..., a_n\}$	Alphabet extension
Composite Processes ($\Vert P$)	
$P \Vert Q$	Parallel composition
forall $[i : 1..n]$ $P(i)$	Replicator construct: equivalent to the parallel composition $(P(1) \Vert ... \Vert P(n))$.
$a : P$	Process labeling

FSP notations and semantics. Table 1 provides an overview of the FSP operators, while the interested reader is referred to [33] for further detail. Briefly stated, FSP processes describe actions (events) that occur in sequence, and choices between event sequences. Each process has an alphabet of the events that it is aware of (and either engages in or refuses to engage in). There are two types of processes: *primitive processes* and *composite processes*. Primitive processes are constructed through action prefix, choice, and sequential composition. Composite processes are constructed using parallel composition or process relabeling. When composed in parallel, processes synchronize on shared events: if processes P and Q are composed in parallel as $P||Q$, events that are in the alphabet of only one of the two processes can occur independently of the other process, but an event that is in the alphabets of both processes cannot occur until the two of them are willing to engage in it. The replicator forall is a convenient syntactic construct used to specify parallel composition over a set of processes. Processes can optionally be parameterized and have re-labeling, hiding or extension over their alphabet. A composite process is distinguished from a primitive process by prefixing its definition with $\Vert$.

A formalization of connectors. According to [1], a connector is defined by a set of *roles* and a *glue* where:

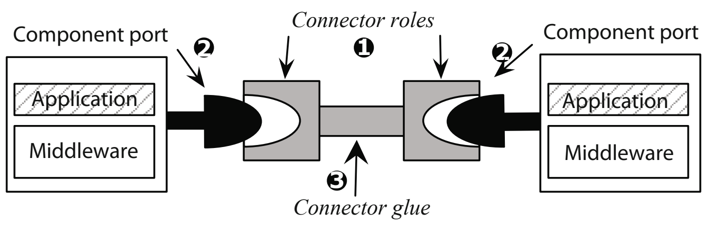


Fig. 2. Components & Connector

– *roles* (See Figure 2, ❶) specify the expected local behavior of each of the interacting parties.
– *glue* (See Figure 2, ❸) specifies how the behaviors of these parties are coordinated.

In addition, the interaction protocols of components are specified by *ports* (See Figure 2, ❷).

Then according to [47], roles, glues and ports are specified as FSP processes, which allows assessing architectural matching and thus interoperability. Specifically, a component can be attached to a connector only if its port is *behaviorally compatible* with the connector role it is bound to. Allen and Garlan [1] define behavioral compatibility between a component port and a connector role based on the notion of refinement. Informally, a component port is behaviorally compatible with a connector role if the process specifying the behavior of the former refines the process characterizing the latter. In other words, it should be possible to substitute the role process by the port process.

In our case, we are further interested in characterizing interaction protocols at both application and middleware layers since both of them are sources of heterogeneity. We then define the behavior of a connector as a hierarchical protocol that specifies the behavior of the application-layer interaction protocol in terms of middleware-specific protocols. Building on the work of [47], the behavior of a middleware-layer connector is specified as a parallel FSP process composing: (i) one process for each role of the connector, and (ii) one process for the glue that describes how all roles are bound together. The application-specific behavior is further specified as a process over role processes of the underlying middleware-layer connector.

Example. As an illustration, we have the following FSP-based specification of a SOAP-based connector:

$$\begin{array}{ll}
\textsf{Role Client}_{SOAP} & = \textit{SOAP-RPCCall} \rightarrow \textit{SOAP-RPCReceiveReply} \rightarrow \textsf{Client}_{SOAP} \\
\textsf{Role Server}_{SOAP} & = \textit{SOAP-RPCReceiveCall} \rightarrow \textit{SOAP-RPCReply} \rightarrow \textsf{Server}_{SOAP} \\
\textsf{Glue}_{SOAP} & = \textit{SOAP-RPCCall} \rightarrow \textit{SOAP-RPCReceiveCall} \rightarrow \textsf{Glue}_{SOAP} \\
 & \mid \textit{SOAP-RPCReply} \rightarrow \textit{SOAP-RPCReceiveReply} \rightarrow \textsf{Glue}_{SOAP} \\
\|\textsf{Connector}_{SOAP} & = \textsf{Client}_{SOAP} \| \textsf{Glue}_{SOAP} \| \textsf{Server}_{SOAP}
\end{array}$$

According to the specification, Client$_{SOAP}$ (Line 1) initiates a request using *SOAP-RPCCall*, and gets a response through *SOAP-RPCReceiveReply*. When Server$_{SOAP}$ (Line 2) gets a request *SOAP-RPCReceiveCall*, it initiates a response *SOAP-RPCReply*. The Glue$_{SOAP}$ coordinates the interaction of the two roles (Lines 3 and 4): a *SOAP-RPCCall* from the Client$_{SOAP}$ is followed by a *SOAP-RPCReceiveCall* to the Server$_{SOAP}$, and a *SOAP-RPCReply* from the Server$_{SOAP}$ is followed by a *SOAP-RPCReceiveReply* to the Client$_{SOAP}$.

Then, different application-layer protocols may be specified using the provided middleware connector. For instance, consider the *Photo Sharing* example discussed in the introduction, Figure 3 gives the FSP specification of the *RPC-SOAP* implementation of infrastructure-based photo sharing. First, we define the SOAP actions that can be performed by the networked systems (Line 1). The behavior of the photo sharing consumer (Lines 3 to 5), producer (Lines 6 to 7), and server (Lines 8 to 12) are specified using this provided set of actions. The photo sharing producer and consumer invoke actions using the Client$_{SOAP}$ process (Lines 14 to 15) while the photo sharing server provides actions using the Server$_{SOAP}$ process(Lines 16 to 17). The Glue$_{SOAP}$ ensures the coordination among all the actions (Lines 18 to 20).

```
 1 //Infrastructure-bade application specification
 2 set SOAP_PhotoSharing_Actions = {uploadPhoto, searchPhoto, downloadPhoto, downloadComment, commentPhoto}
 3 PhotoSharingConsumer          = (req.searchPhoto →P1),
 4 P1                            = (req.downloadPhoto →P1|req.commentPhoto →P1
 5                                 |req.downloadComment →P1 |terminate →END).
 6 PhotoSharingProducer          = (req.uploadPhoto →PhotoSharingProducer
 7                                 |terminate →END).
 8 PhotoSharingServer            = (prov.uploadPhoto →PhotoSharingServer
 9                                 |prov.searchPhoto →PhotoSharingServer
10                                 |prov.downloadPhoto →PhotoSharingServer
11                                 |prov.commentPhoto →PhotoSharingServer
12                                 |prov.downloadComment →PhotoSharingServer|terminate →END).
13 //SOAP middleware Specification
14 Client_SOAP(X =' op)          = (req.[X] →P1|terminate →END),
15 P1                            = (SOAP-RPCCall[X] → SOAP-RPCReceiveReply[X] →Client_SOAP).
16 Server_SOAP(X =' op)          = (prov.[X] →P2 |terminate →END),
17 P2                            = (SOAP-RPCReceiveCall[X] → SOAP-RPCReply[X] →Server_SOAP).
18 Glue_SOAP(X =' op)            = (SOAP-RPCCall[X] →P0 |terminate →END),
19 P0                            = (SOAP-RPCReceiveCall[X] → SOAP-RPCReply[X]
20                                 → SOAP-RPCReceiveReply[X] →Glue_SOAP).
21 //System specification
22 ||SOAP_PhotoSharing           = (PhotoSharingProducer
23                                 ||PhotoSharingConsumer
24                                 ||PhotoSharingServer
25                                 ||(forall [op:SOAP_PhotoSharing_Actions] Server_SOAP(op))
26                                 ||(forall [op:SOAP_PhotoSharing_Actions] Client_SOAP(op))
27                                 ||(forall [op:SOAP_PhotoSharing_Actions] Glue_SOAP(op))).
```

Fig. 3. Infrastructure-based photo sharing

2.3 Connection Mismatches and Related Mediation

Connection mismatches result from different assumptions that components make about connection. Blair et al. [6] define several heterogeneity dimensions generating mismatches (see Figure 4):

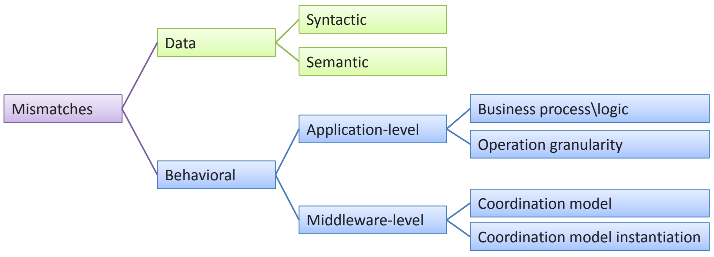


Fig. 4. Classifying mismatches

– *Data heterogeneity:* Networked systems associate different representations (syntax) and meanings (semantics) to their data, which may results in data inconsistencies. Middleware coupled with ontologies play a valuable role in solving both the syntactic and semantic mismatches.
– *Behavioral middleware-level heterogeneity:* While middleware ensures interoperability across languages and network platforms, it only does so for systems using the same middleware. Indeed, a middleware implementation involves a style of interaction by specifying a coordination model and the associated protocol and data format. As a result, systems using different middleware are not able to interoperate.
– *Behavioral application-level heterogeneity:* Different systems may have incompatible business-process logic and disparate interface signatures (e.g., see [26]).

A further dimension of heterogeneity is related to the handling of non-functional properties, which we do not address in this chapter.

In general, networked systems may be connected only in the absence of all of the above heterogeneity dimensions, i.e., networked systems should be behaviorally compatible from application down to middleware layer, and further exchange semantically and syntactically matching data.

However, with networked systems getting increasingly pervasive, one would like to be able to connect networked systems that *semantically match*, despite heterogeneity in the above dimensions. By *semantic matching* [41], we mean that networked systems share a complementary high level goal towards which they need to coordinate although they may possibly run heterogeneous interaction protocols from the application down to the middleware layer.

Still considering the photo sharing example, both the infrastructure-based (see Figure 5A) and the peer-to-peer-based (see Figure 5B) versions of the photo sharing may be implemented over SOAP. Even though, the two systems semantically match and behaviorally match at the middleware-layer, they are not able to interact due to behavioral mismatches at the application layer.

Similarly, the infrastructure-based version of photo sharing may be implemented using two different middleware, such as SOAP (see Figure 6A) and RMI(see Figure 6B). In this case, middleware-layer mismatches prevent the two

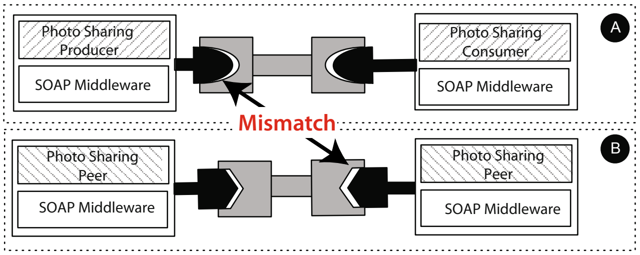


Fig. 5. Application mismatches

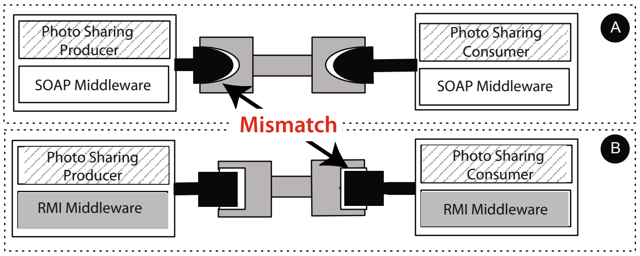


Fig. 6. Middleware mismatches

networked systems from interacting despite semantic matching and behavioral matching at the application-layer.

Under semantic matching of two networked systems, behavioral matchmaking is achieved through the generation of *mediators* that enforce the behavioral compatibility of the networked systems (see Figure 7). The resulting system is called *connected system*.

Since mismatches take place at different inter-related layers, mediation becomes a cross-cutting concern that has to be achieved in conjunction at the different system layers, from application down to middleware down to network (see Figure 8). At each layer, many facets (data, interface, and behavior) of heterogeneity should be dealt with. There is a number of existing mediation solutions, each of which solves mismatches related either to applications or to middleware. Indeed, solutions addressing application heterogeneity assume the same middleware whereas solutions achieving middleware interoperability consider the same application atop of it. However, all the dimensions of heterogeneity should be simultaneously addressed in order to guarantee effective interoperability among heterogeneous systems.

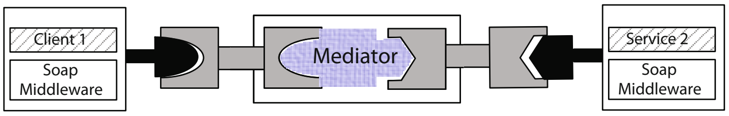


Fig. 7. Connected system

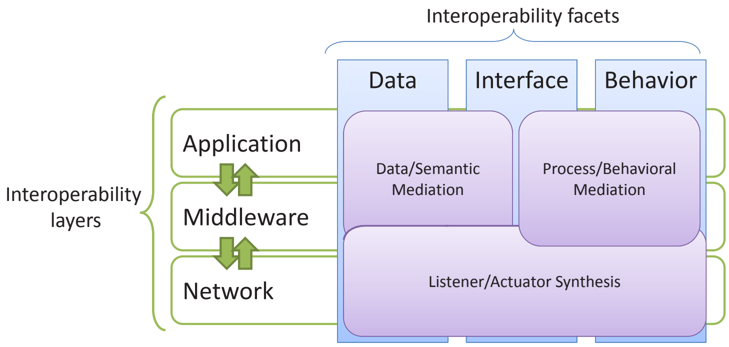


Fig. 8. Mediation

In this chapter, we more specifically concentrate on middleware-layer protocol mediation and its relation with application-layer mediation. As a first step, the next section reviews state-of-the-art solutions to middleware interoperability that is in particular surveyed in [6]. We qualify such solutions as *interoperability connectors*. However, these solutions primarily deal with middleware-level heterogeneity, further assuming connection between components relying on the same interaction paradigm. Section 4 then introduces the solution investigated within the CONNECT project that aims at overcoming both application- and middleware-level heterogeneity, including heterogeneity in the interaction paradigms.

3 Interoperability Connectors

State-of-the-art solutions to interoperability between heterogeneous middleware primarily concentrate on middleware implementing the same interaction paradigm and subdivide into the following categories: *software bridge*, *interoperability platform* and *transparent interoperability* [6]. We review each approach in turn, providing their FSP-based semantics so as to precisely characterize their respective features and further allow for thorough assessment and comparison. In addition, we point out exploitation of the proposed interoperability connectors at the application layer.

3.1 Software Bridges

Bridging assumes *a priori* knowledge of both applications and middleware that have to be made interoperable without any intervention in their code. Particularly, bridging provides a mapping between various interaction protocols. Such a mapping can be either $1 \rightarrow 1$, which is *direct bridging*; or $n \rightarrow 1 \rightarrow m$, which is *indirect bridging*.

Direct Bridging. The principle of *direct bridging* is to transform one of the connector roles according the incompatible connector role, as illustrated in Figure 9.

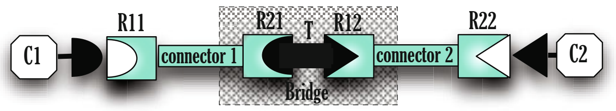


Fig. 9. Direct bridge

Formally, direct bridging is performed as follows (see Figure 10):

1. The glue of each connector is first tagged in order to avoid unwanted event synchronization (tag_1 :Glue_1 and tag_2 :Glue_2, Line 11),
2. A set of predefined transformations, T (Line 7), is applied to the connectors in order to adapt their respective behaviors,
3. The transformations are chained with the tagged glues through the Bridge (Line 5) process.

According to the above, the *direct bridge mediator* specification is defined as:

$$||\mathsf{Direct\_Bridge\_Mediator} = (\mathsf{Bridge}||\mathsf{T}).$$

The developer must thus ensure the correctness of T, in particular ensuring that the bridging actually performs the required mediation without introducing any error. Moreover, a direct bridge must be developed separately for every pair of protocols between which interaction is required. Hence, ensuring interoperability between each pair of n components requires developing $n(n-1)$ mediators. The diversity of protocols that are used in today's networked systems implies that this is a substantial development task.

```
  //Specification of Connector$_1$ & Connector$_2$
1 Role R1$_{i,i\in[1..2]}$  = Specification of Role R1 of Connector$_i$
2 Role R2$_{i,i\in[1..2]}$  = Specification of Role R2 of Connector$_i$
3 Glue$_{i,i\in[1..2]}$     = Specification of the glue of Connector$_i$
4 set I$_{i,i\in[1..2]}$    = Set of events initiated from  Role R1$_i$ and R2$_i$
5 Bridge             = $tag_1.[e_1 : I_1] \rightarrow tag_2.[e_1] \rightarrow$Bridge|$tag_2.[e_2 : I_2] \rightarrow tag_1.[e_2] \rightarrow$Bridge

6 //Specification of the adaptation process
7 T                  = Specification of the required transformations to bridge Connector$_1$ to Connector$_2$

8 //Specification of the direct bridge connector
9 ||C-DBridge        = R1$_1$||$tag_1$ :Glue$_1$||Bridge ||T||$tag_2$ :Glue$_2$||$R2_2$
```

Fig. 10. Direct bridging specification

Practically, middleware direct bridges, such as OrbixCOMet[4] and SOAP2-CORBA[5], ensure interoperability between two fixed middleware implementations (DCOM-CORBA and SOAP-CORBA respectively). Similarly, application software bridges may be introduced to define bridging between application-specific protocols (i.e., overcoming application- and middleware-layer protocols heterogeneity). However, implementing a bridge between two networked applications becomes very complex due to the domain-specific and technical knowledge required to realize the mediation.

Indirect Bridging. Indirect bridging reduces the development effort associated with software bridges by introducing a common fixed intermediary protocol. This intermediary protocol is represented as a dedicated connector called $\texttt{Connector}_{bus}$ (see Figure 11). Then, interoperability is achieved in two steps: (i) the given native middleware protocol taken among n middleware is translated into a common intermediary protocol, (ii) the common intermediary protocol is then translated into the other given native middleware protocol taken among m middleware.

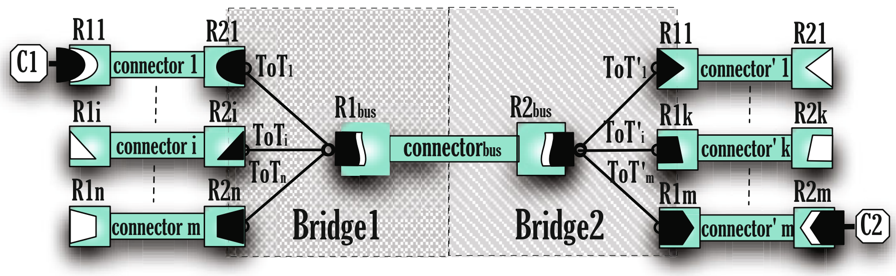


Fig. 11. Indirect bridge

Formally, indirect bridging performs translations back and forth using direct bridges in two steps (see Figure 12):

1. $\texttt{Connector}_i$ to $\texttt{Connector}_{bus}$ direct bridging through the use of processes $ToT_i \| \mathsf{Bridge}_i$ (Lines 19 and 26) ($i \in [1..n]$),
2. $\texttt{Connector}_{bus}$ to $\texttt{Connector}'_k$ direct bridging through the use of processes $ToT'_k \| \mathsf{Bridge}'_k$ (lines 22 and 29) ($k \in [1..m]$).

The *indirect bridge mediator* is then specified as:

$$\|\mathsf{Indirect\_Bridge\_Mediator} = (\mathsf{T}_1 \| \mathsf{Bridge}_1 \| \mathsf{Bridge}_2 \| \mathsf{T}_2).$$

Practically, there exist various implementations of indirect bridges such as Enterprise Service Buses (e.g., ARTIX[6]) and MUSDAC [43]. Especially, Enterprise

[4] http://www.iona.com/support/whitepapers/ocomet-wp
[5] http://soap2corba.sourceforge.net/
[6] http://web.progress.com/en/sonic/artix-index.html

```
//$\mathtt{Connector}_{bus}$ specification
Role $\mathsf{R1}_{bus}$         = Specification of Role R1 of $\mathtt{Connector}_{bus}$
Role $\mathsf{R2}_{bus}$         = Specification of Role R2 of $\mathtt{Connector}_{bus}$
$\mathsf{Glue}_{bus}$            = Specification that describes interactions between Role $\mathsf{R1}_{bus}$ and Role $\mathsf{R2}_{bus}$

//Connectors specification
Role R1              = $|_{i=1}^{n}(a.glue_i \rightarrow \mathsf{R1}_i)$,
$\mathsf{R1}_{i,i\in[1\cdots n]}$       = $\mathsf{R1}_i$ initial specification as given by $\mathtt{Connector}_i|reset \rightarrow \mathsf{R1}$
Role R2              = $|_{k=1}^{m}(b.glue'_k \rightarrow \mathsf{R2}_k)$,
$\mathsf{R2}_{k,k\in[1\cdots m]}$       = $\mathsf{R2}_k$ initial specification as given by $\mathtt{Connector'}_k|reset \rightarrow \mathsf{R2}$
$\mathsf{Glue}_{i,i\in[1\cdots n]}$     = Specification that describes interactions between
                       Roles $\mathsf{R1}_i$ and $\mathsf{R2}_i$
$\mathsf{Glue'}_{k,k\in[1\cdots m]}$    = Specification that describes interactions between
                       Role $\mathsf{R'1}_k$ and Role $\mathsf{R'2}_k$

//Set of events initiated or observed
set $\mathsf{I1}_{i,i\in[1\cdots n]}$   = Set of events initiated from Role $\mathsf{R1}_i$
set $\mathsf{O1}_{i,i\in[1\cdots n]}$   = Set of events observed from Role $\mathsf{R1}_i$
set $\mathsf{I2}_{k,k\in[1\cdots m]}$   = Set of events initiated from Role $\mathsf{R'2}_k$
set $\mathsf{O2}_{k,k\in[1\cdots m]}$   = Set of events observed from Role $\mathsf{R'2}_k$

//Specification of the adaptation processes
$\mathsf{T}_1$                 = $|_{i=1}^{n}(a.glue_i \rightarrow \mathsf{ToT}_i)$,
$\mathsf{ToT}_{i,i\in[1\cdots n]}$      = Specification of the required transformations to bridge $\mathtt{Connector}_i$ to $\mathtt{Connector}_{bus}$
                     | $a.reset \rightarrow \mathsf{T}_1$
$\mathsf{T}_2$                 = $|_{k=1}^{m}(b.glue'_k \rightarrow \mathsf{ToT}'_k)$,
$\mathsf{ToT}'_{k,k\in[1\cdots m]}$     = Specification of the required transformations to bridge$\mathtt{Connector}_{bus}$ to $\mathtt{Connector}'_k$
                     | $b.reset \rightarrow \mathsf{T}_2$

//Specification of the bridging processes
$\mathsf{Bridge}_1$            = $|_{i=1}^{n}(a.glue_i \rightarrow \mathsf{Bridge}_i)$,
$\mathsf{Bridge}_{i,i\in[1\cdots n]}$   = $[e : \mathsf{I1}_i] \rightarrow a.tag_i.[e] \rightarrow \mathsf{Bridge}_i | a.tag_i.[e : \mathsf{O1}_i] \rightarrow [e] \rightarrow \mathsf{Bridge}_i$
                     | $a.reset \rightarrow \mathsf{Bridge}_1$
$\mathsf{Bridge}_2$            = $|_{k=1}^{m}(b.glue'_k \rightarrow \mathsf{Bridge}'_k)$,
$\mathsf{Bridge}'_{k,k\in[1\cdots m]}$  = $[e : \mathsf{I2}_k] \rightarrow b.tag_k.[e] \rightarrow \mathsf{Bridge}'_k | b.tag_i.[e : \mathsf{O2}_k] \rightarrow [e] \rightarrow \mathsf{Bridge}'_k$
                     | $b.reset \rightarrow \mathsf{Bridge}_2$

//Specification of the indirect bridge connector
$\|$C-IBridge          = $\mathsf{R1}\|\mathsf{T}_1\|_{i=1}^{n} a.tag_i : \mathsf{Glue}_i\|\mathsf{Bridge}_1\|\mathsf{Glue}_{bus}\|\mathsf{Bridge}_2\|_{k=1}^{m} b.tag_k : \mathsf{Glue}'_k\|\mathsf{T}_2\|\mathsf{R2}$
```

Fig. 12. Indirect bridging specification

Service Buses (ESBs) have received a lot of attention. An ESB [35] is an open standards, message-based, distributed integration infrastructure that provides routing, invocation and mediation services to facilitate the interactions of disparate distributed applications and services.

Compared to direct bridging that requires $n \times m$ direct bridges to allow n components to interact with m, indirect bridging reduces the development effort since $n + m$ bridges have to be manually developed. Nevertheless, it limits the expressiveness of protocols, as some aspects of the relevant protocols may not be compatible with the chosen intermediary protocol.

3.2 Interoperability Platform

To overcome the static nature of software bridging, new approaches that dynamically select the best middleware bridge at a given time and place have emerged. Such solutions, called thereafter *interoperability platforms*, enable networked systems to switch their interaction protocol on-the-fly according to their environment. The principle is to provide a custom interface that abstracts the different interaction protocols used in the environment (see Figure 13).

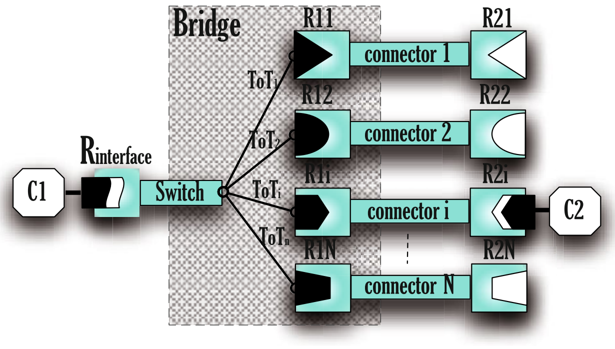


Fig. 13. Interoperability platform

Formally, interoperability is ensured in the following steps (see Figure 14):

1. The common interface, that has to be used by any component to interoperate with its environment is formally specified by a role $R_{interface}$ (Line 2),

```
1  //Proprietary interface
2  Role $R_{interface}$          $=$ Specification of the bridge interface
3  Role R2                     $= |_{i=1}^{n}(glue_i \rightarrow \mathsf{R2}_i)$,
4  $\mathsf{R2}_{i,i\in[1..n]}$              $=$ Initial specification of the Role R2 of Connector$_i$ $| reset \rightarrow$ R2
5  $\mathsf{Glue}_{i,i\in[1..n]}$            $=$ Specification of the glue of Connector$_i$
6  //Set of events initiated or observed
7  set $\mathsf{I2}_{i,i\in[1..n]}$          $=$ Set of events initiated from Role R2$_i$
8  set $\mathsf{O2}_{i,i\in[1..n]}$          $=$ Set of events observed from Role R2$_i$
9  set $\mathsf{I}_{interface}$          $=$ Set of events initiated from Role R$_{interface}$
10 set $\mathsf{O}_{interface}$          $=$ Set of events observed from Role R$_{interface}$

11 //Switch process
12 Switch                      $= (election \rightarrow reset \rightarrow \mathsf{Switch}\ |_{i=1}^{n} election \rightarrow glue_i \rightarrow \mathsf{Switch}) \backslash \{election\}$

13 //Specification of the adaptation process
14 T                           $= |_{i=1}^{n}(glue_i \rightarrow \mathsf{ToT}_i)$,
15 $\mathsf{ToT}_{i,i\in[1..n]}$             $=$ Specification of the required transformations to bridge R$_{interface}$ to Connector$_i$
16                             $| \ reset \rightarrow \mathsf{T}$

17 //Specification of the bridging process
18 Bridge                      $= |_{i=1}^{n}(glue_i \rightarrow \mathsf{Bridge}_i)$,
19 $\mathsf{Bridge}_{i,i\in[1..n]}$          $= [e : \mathsf{R}_{interface}] \rightarrow tag_i.[e] \rightarrow \mathsf{Bridge}_i | tag_i.[e : \mathsf{O}_{interface}] \rightarrow [e] \rightarrow \mathsf{Bridge}_i$
20                             $| \ [e : \mathsf{I2}_i] \rightarrow tag_i.[e] \rightarrow \mathsf{Bridge}_i | tag_i.[e : \mathsf{O2}_i] \rightarrow [e] \rightarrow \mathsf{Bridge}_i$
21                             $| \ reset \rightarrow \mathsf{Bridge}$

22 //Specification of the interoperability platform connector
23 ||C-InteropPlatforms $= \mathsf{R}_{interface} || \mathsf{Switch} || \mathsf{T} || \mathsf{Bridge} ||_{i=1}^{n} tag_i : \mathsf{Glue}_i || \mathsf{R}_2$
```

Fig. 14. Interoperability platform specification

2. The Switch process (Line 12) selects the appropriate connector $\texttt{Connector}_i$ among n according to the requirements of the environment,
3. The translation between $R_{interface}$ and $\texttt{Connector}_i$ is achieved in a way similar to direct bridging (Lines 14 to 21).

This leads to the following specification of the *interoperability mediator*:

$$||\mathsf{Interoperability\_Mediator} = (\mathsf{Switch}||\mathsf{T}||\mathsf{Bridge}).$$

Practically, middleware-level interoperability platforms, such as UIC [44] and ReMMoC [21], allow the development of applications independently from the underlying protocol. They select the most appropriate communication protocol according to the context. Many applications, however, have not been developed using such middleware interface and cannot be modified because their source codes are not available. From the perspective of application-layer protocols, the common interface is in general a domain-specific standard that several components and services comply with. However, compliance to the same interface does not necessarily imply behavioral compatibility and mediators have to be used in order to guarantee behavioral compatibility as well [15].

3.3 Transparent Interoperability

Unlike indirect bridging, transparent interoperability solutions do not rely on a fixed common protocol anymore but rather synthesize the common protocol dynamically based on the interaction behavior of communicating parties. We are more specifically interested in *dynamic protocol translation* [7]. This approach is based on concepts taken from the theory of protocol projection [30]. The theory enables mapping incompatible protocols to an image protocol (see Figure 15), which has proven effective to reason about conversions and semantic equivalence among heterogeneous protocols [7]. In particular, an image protocol abstracts incompatibilities among protocols to exclusively consider their similarities. Further, by generating an image protocol on-the-fly, it is possible to provide a dynamic semantic correspondence among heterogeneous middleware protocols.

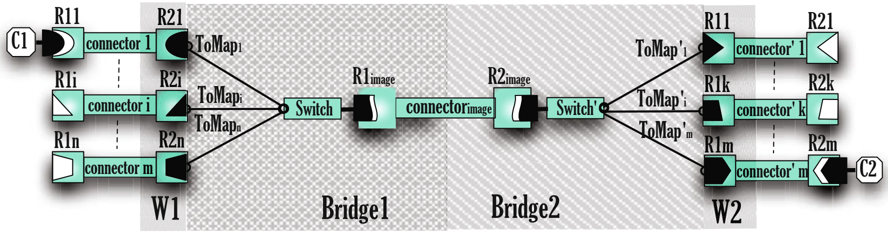


Fig. 15. Transparent interoperability

Formally, a projection function f is used to synthesize an image protocol, that is the greatest common denominator between a pair of protocols (see Figure 16). Interoperability is then performed in the following steps:

1. The glue of all the connectors are tagged in order to avoid unwanted event synchronization,
2. One connector is dynamically chosen among n (m) connectors based on the context/environment through the Switch (Switch') process: $\mathtt{Connector}_i$ ($\mathtt{Connector}'_k$) (Lines 19 and 20),
3. W_1 (W_2) (Lines 22 to 25) are then used to synchronize tagged glues with their respective roles depending on the selected connector,
4. The strength of the approach lies in M_1 and M_2 processes (Lines 26 to 32) that are used to define the semantics of the events. To do so, the projection function (f) is used to establish the semantic equivalence between events: $f(e_1) = f(e_2)$ iff e_1 and e_2 have the same semantics,
5. Bridge_1 and Bridge_2 (Lines 34 to 41) tag/untag the projected events in order to allow M_1 and M_2 to synchronize.

This leads to the following specification of the *transparent mediator*:

$$\|\mathsf{Transparent\_Mediator} = (\mathsf{Switch}\ \|\mathsf{W}\ _1\|\mathsf{M}\ _1\|\mathsf{Bridge}_1\|\mathsf{Bridge}_2\|\mathsf{M}\ _2\|\ \mathsf{W}_2\|\mathsf{Switch}').$$

Practically, the INDISS [9] and NEMESYS [7] middleware implement the dynamic protocol translation approach for service discovery and interaction protocol (assuming the same application atop) respectively. uMiddle [36], OSDA [31], SeDiM [19] are other middleware-level implementations of the transparent interoperability approach. Regarding the application layer, there is a substantial piece of work on transparent interoperability at the application layer assuming the use of Semantic Web technologies. OWL-S [51] exploit Semantic Web ontologies to enrich descriptions of services in order to enhance service discovery and composition using semantic matching. Web Service Modeling Ontology (WSMO)[7] introduces mediators as the core of a conceptual model treating heterogeneity of Semantic Web Services. In particular, it addresses both data and behavioral mediation.

As briefly surveyed in this section, tremendous work exists on the development of concrete interoperability solutions to overcome protocol heterogeneity and in particular middleware protocol heterogeneity. However, these solutions focus on a single protocol layer, while the connection of pervasive networked systems requires dealing with protocol heterogeneity at both application and middleware layers. In addition, middleware heterogeneity is in general overcome for middleware protocols implementing the same interaction paradigms while the increasing heterogeneity of the networked devices now calls for connecting systems relying on different interaction paradigms.

[7] http://www.wsmo.org/

```
$//\texttt{Connectors}\ \textit{specification}$
$\mathsf{Role\ R1}$                 $= |_{i=1}^{n}(a.glue_i \rightarrow \mathsf{R1}_i),$
$\mathsf{R1}_{i,i\in[1\cdots n]}$   $= \mathsf{R1}_i\ \textit{Initial specification as given by}\ \texttt{Connector}_i|reset \rightarrow \mathsf{R1}$
$\mathsf{Role}\ R2$                 $= |_{k=1}^{n}(b.glue_k \rightarrow \mathsf{R2}_k),$
$\mathsf{R2}_{k,k\in[1\cdots n]}$   $= \mathsf{R2}_k\ \textit{Initial specification as given by}\ \texttt{Connector}'_k|reset \rightarrow \mathsf{R2}$
$\mathsf{Glue}_{i,i\in[1\cdots n]}$ $= \textit{Specification that describes interactions between}\ \mathsf{Role\ R1}_i\ \mathsf{and\ Role\ R2}_i$
$\mathsf{Glue'}_{k,k\in[1\cdots m]}$ $= \textit{specification that describes interactions between}\ \mathsf{Role\ R'1}_k\ \mathsf{and\ Role\ R'2}_k$

$//\textit{Definition of set of events}$
$\mathsf{set\ I1}_{i,i\in[1\cdots n]}$ $= \textit{Set of events initiated from}\ \ \mathsf{Role\ R1}_i$
$\mathsf{set\ O1}_{i,i\in[1\cdots n]}$ $= \textit{Set of events observed from}\ \ \mathsf{Role\ R1}_i$
$\mathsf{set\ I2}_{k,k\in[1\cdots m]}$ $= \textit{Set of events initiated from}\ \ \mathsf{Role}\ R'2_k$
$\mathsf{set\ O2}_{k,k\in[1\cdots m]}$ $= \textit{Set of events observed from}\ \ \mathsf{Role}\ R'2_k$
$\mathsf{set\ E}_{i,i\in[1\cdots n]}$  $= \alpha\mathrm{R1}_i \cap \alpha\mathsf{Glue}_i$
$\mathsf{set\ E}_{k,k\in[1\cdots m]}$  $= \alpha\mathrm{R2}_k \cap \alpha\mathsf{Glue'}_k$
$\mathsf{set}\ \sum_{\mathsf{E1}_n}$   $= \cup_{i=1}^{n}\mathsf{E1}_i$
$\mathsf{set}\ \sum_{\mathsf{E2}_m}$   $= \cup_{k=1}^{m}\mathsf{E2}_k$
$\mathsf{set}\ \sum_{\mathsf{O1}_n}$   $= \cup_{i=1}^{n}\mathsf{O1}_i$
$\mathsf{set}\ \sum_{\mathsf{O2}_m}$   $= \cup_{k=1}^{m}\mathsf{O2}_k$
$\mathsf{Switch}$                      $= (a.election \rightarrow a.reset \rightarrow \mathsf{Switch}|_{i=1}^{n}a.election \rightarrow a.glue_i \rightarrow \mathsf{Switch})\backslash\{a.election\}$
$\mathsf{Switch'}$                     $= (b.election \rightarrow b.reset \rightarrow \mathsf{Switch'}|_{k=1}^{m}b.election \rightarrow b.glue'_k \rightarrow \mathsf{Switch'})\backslash\{b.election\}$

$//\textit{Specification of processes for the image protocol generation}$
$\mathsf{W}_1$                              $= |_{i=1}^{n}(a.glue_i \rightarrow ToGlue_i),$
$\mathsf{ToGlue}_{i,i\in[1\cdots n]}$       $= [e : \mathsf{I1}_i] \rightarrow a.tag_i.[e] \rightarrow \mathsf{ToGlue}_i\ |a.tag_i.[e : \mathsf{O1}_i] \rightarrow [e] \rightarrow \mathsf{ToGlue}_i\ |a.reset \rightarrow \mathsf{W}_1$
$\mathsf{W}_2$                              $= |_{k=1}^{n}(b.glue'_k \rightarrow \mathsf{ToGlue}'_k),$
$\mathsf{ToGlue}'_{k,k\in[1\cdots m]}$      $= [e : \mathsf{I2}_k] \rightarrow b.tag_k.[e] \rightarrow \mathsf{ToGlue}'_k\ |b.tag_k.[e : \mathsf{O2}_k] \rightarrow [e] \rightarrow \mathsf{ToGlue}'_k\ |b.reset \rightarrow \mathsf{W}_2$
$\mathsf{M}_1$                              $= |_{i=1}^{n}(a.glue_i \rightarrow \mathsf{ToMap}_i),$
$\mathsf{ToMap}_{i,i\in[1\cdots n]}$        $= a.tag_i.[e : \mathsf{I1}_i] \rightarrow a.tag_i.f(e) \rightarrow \mathsf{ToMap}_i$
                                            $|\ \ a.tag_i.f(e : \sum_{\mathsf{O1}_n}) \rightarrow a.tag_i.[e : \mathsf{O1}_i] \rightarrow \mathsf{ToMap}_i|a.reset \rightarrow \mathsf{M}_1$
$\mathsf{M}_2$                              $= |_{k=1}^{n}(b.glue'_k \rightarrow \mathsf{ToMap}'_k),$
$\mathsf{ToMap}'_{k,k\in[1\cdots m]}$       $= b.tag_k.[e : \mathsf{I2}_k] \rightarrow b.tag_k.f(e) \rightarrow \mathsf{ToMap}'_k$
                                            $|\ \ b.tag_k.f(e : \sum_{\mathsf{O2}_m}) \rightarrow b.tag_k.[e : \mathsf{O2}_k] \rightarrow \mathsf{ToMap}'_k$
                                            $|\ \ b.reset \rightarrow \mathsf{M}_2$

$//\textit{Specification of the bridging processes}$
$\mathsf{Bridge}_1$                         $= |_{i=1}^{n}(a.glue_i \rightarrow \mathsf{ToBridge}_i),$
$\mathsf{ToBridge}_{i,i\in[1\cdots n]}$     $= a.tag_i.f\left(e_2 : \sum_{\mathsf{E2}_k}\right) \rightarrow f(e_2) \rightarrow \mathsf{ToBridge}_i$
                                            $|\ \ f(e_1 : \sum_{\mathsf{E1}_n}) \rightarrow a.tag_i.f(e_1) \rightarrow \mathsf{ToBridge}_i$
                                            $|\ \ a.reset \rightarrow \mathsf{Bridge}_1$
$\mathsf{Bridge}_2$                         $= |_{k=1}^{m}(b.glue'_k \rightarrow \mathsf{ToBridge}'_k),$
$\mathsf{ToBridge}'_{k,k\in[1\cdots m]}$    $= b.tag_k.f\left(e_1 : \sum_{E1_n}\right) \rightarrow f(e_1) \rightarrow \mathsf{ToBridge}'_k$
                                            $|\ \ f(e_2 : \sum_{\mathsf{E2}_m}) \rightarrow b.tag_k.f(e_2) \rightarrow \mathsf{ToBridge}'_k$
                                            $|\ \ b.reset \rightarrow \mathsf{Bridge}_2$

$//\textit{Specification of the transparent interoperability connector}$
$||\mathsf{C\text{-}Transparent\_Interop} = \mathsf{R1}||\ \mathsf{Switch}\ ||_{i=1}^{n}a.tag_i : \mathsf{Glue}_i/\{f(r : \alpha\mathsf{Glue}_i)/[r]\}||\ \mathsf{W}\ _1||\ \mathsf{M}\ _1||\mathsf{Bridge}_1||\mathsf{Bridge}_2$
                                            $||\mathsf{M}\ _2||\ \mathsf{W}\ _2||_{k=1}^{m}b.tag_k : \mathsf{Glue}_k/\{f(r : \alpha\mathsf{Glue}_k)/[r]\}||\mathsf{Switch}\ '||\mathsf{R2}$
```

Fig. 16. Transparent interoperability specification

4 Emergent Connector Synthesis

Towards overcoming the increasing heterogeneity of today's pervasive networking environments, this section introduces a model-based approach to the synthesis of emergent connectors, which builds upon the theory of mediators introduced for application-layer protocols in [46] and further surveyed in companion chapter

[26]. An emergent connector allows two networked systems that complementary provide/require the same functionality to coordinate although they possibly execute different protocols. This then requires adequate modeling of networked systems to enable reasoning about their semantic and behavioral compatibility/matching (Section 4.1), which in particular relies on the definition of ontologies conceptualizing middleware and application functions (Section 4.2). Briefly stated, two networked systems are considered to be *semantically matching* if they respectively require and provide a matching high-level functionality, which is characterized by ontology concepts. Then, assessing whether the two networked systems are *behaviorally compatible* relies on analyzing whether the protocols associated with the realization of the given functionality may be adapted so that they can successfully coordinate. The resulting adaptation then defines the mediator to be implemented by the emergent connector. As illustrated by the rich literature on protocol conversion (e.g., [11]), different compatibility relations may be defined. They primarily differ according to their complexity and conversely proportional flexibility. In order to lower the complexity of emergent connectors, we perform protocol mediation according to known mapping between the networked systems' actions, which is inferred from their ontology-based semantics. In addition, protocol mediation is composed according to the basic mediation patterns known from the literature (Section 4.3), while concrete connectors handle actual middleware message translation (Section 4.4). Finally, our work takes inspiration from extensive literature in the area of protocol mediation and middleware interoperability; our contribution primarily lies in dealing with mediation from application down to the middleware layer (Section 4.5).

4.1 Modeling Networked Systems towards On-the-Fly Connection

A basic assumption of on-the-fly connection of networked systems is that systems advertise their presence in the network(s) they join. This is now common in pervasive networks and supported by a number of resource discovery protocols [53]. Still, a crucial question is which description of resources should be advertised, which ranges from simple (attribute, value) pairs as with SLP[8] to advanced ontology-based interface specification [3].

In our work, resource description shall enable networked systems to compose according to the high-level functionalities they provide and/or require in the network, despite heterogeneity in the protocols associated with the implementation of the functionality. In other words, networked systems must advertise the *high-level functionalities* they provide and/or consume to be able to meet according to the matching of their respective functionalities. Building upon Semantic Web Services, we call such functionalities *capabilities* and we say that networked systems *semantically match* when a networked system requires a capability that matches a capability provided by the other. Then, in accordance with the definition of connectors discussed in Section 2, connection between semantically matching networked systems requires precise characterization of the

[8] http://www.openslp.org/

protocols associated with the realization of capabilities, where protocols are defined as processes over the networked system's observable actions. Observable actions are typically specified as part of the system's *interface signature* while the modeling of protocols relies on some concurrent language and may be advertised by the system or be possibly learned. Last but not least, the semantics of observable actions need to be rigorously defined in order to assign the same meaning to actions in any environment, for which we exploit ontologies.

The following paragraphs further define the notions of *capability*, *interface signature*, and c*capability protocol*.

Capability. Using the terminology of the Semantic Web Services area[9], a *capability* denotes a high-level functionality provided or required from the networked environment. Concretely, a capability is specified as a tuple:

$$Capability =< Type, \mathcal{C}, I, O >$$

where:

- *Type* stands for required (noted *Req*), provided (noted *Prov*) or required and provided (noted *Req_Prov*) capability. A provided capability denotes a capability offered in the network while a required one is to be consumed. A required and provided capability is then both consumed and offered by the networked system, as common in peer-to-peer systems.
- $\mathcal{C}$ gives the semantics of the capability in terms of an ontology concept;
- I (resp. O) specifies the set of inputs (resp. outputs) of the capability, which is defined as a tuple $< i_1, ..., i_n >$ (resp. $< o_1, ..., o_m >$) with $i_{l=[1..n]}$ (resp. $o_{l=[1..m]}$) being an ontology concept.

and where the ontology concepts are defined by a domain-specific ontology referred to in the networked system's interface. As an illustration, the capability of the photo sharing consumer application is defined as:

$$<Req,\ Photo\text{-}Sharing\_Consumer,\ Comment,\ Photo>$$

where the meaning of concepts is direct from the given names (see further Section 4.2 for the definition of the ontology).

Interface Signature. The interface signature of a networked system specifies the set of observable actions that the system executes to interact with other systems. In particular, networked systems implement advertised capabilities as protocols over observable actions that are defined in their interfaces. Usually, the interface signature abstracts the specific middleware functions that the system calls to carry out actions in the network. However, this is due to the fact that existing interface definition languages are closely tight to a specific middleware solution, while we target pervasive networking environments hosting heterogeneous middleware solutions. The specification of an action should then

[9] http://www.ai.sri.com/daml/services/owl-s/

be enriched with the one of the middleware function that is specifically used to carry out that action; indeed, an observable action in an open pervasive network is the conjunction of an application-layer with a middleware-layer function. Middleware functions then need to be unambiguously characterized, which leads us to introduce a middleware ontology that defines key concepts associated with state-of-the-art middleware API, as presented in the next section.

Given the above, the interface of a networked system is defined as a set of actions where each action is described as a tuple: $< m_f, a, I, O >$, where: m_f denotes a middleware function; a denotes the application action; I (resp. O) denotes the set of inputs (resp. outputs) of the action. Moreover, as detailed in Section 4.2, the tuple elements are ontology concepts so that their semantics may be reasoned upon.

As an illustration, Figure 17[10] gives the interface signatures associated with the infrastructure-based implementation of photo sharing. The interfaces refer to ontology concepts from the middleware and application-specific domains of the target scenario; however, this does not prevent general understanding of the signatures given the self-explanatory naming of concepts. Three interface signatures are introduced, which are respectively associated with the *producer*, *consumer* and *server* networked systems. The definition of the systems' actions specify the associated SOAP functions, i.e., the client-side application actions are invoked though SOAP middleware using the *SOAP-RPCCall* function followed by the *SOAP-RPCReceiveReply* function, while they are processed on the server side using the two functions *SOAP-RPCReceiveCall* and *SOAP-RPCReply*. The specific applications actions are rather straightforward from the informal sketch of the scenario introduced in Section 1. For instance, the producer invokes the server operations *Authenticate* and *UploadPhoto* for authentication and photo upload, respectively. The consumer may possibly search for, download or comment photos, or download comments. Finally, the actions of the photo sharing server are complementary to the client actions.

Unlike the infrastructure-based implementation, the peer-to-peer-based photo sharing defines a single interface signature (see Figure 18), as all the peers feature the same capability. The interface further illustrates the naming of actions after domain data types of the application data instead of operations since the actions are data-centric and are performed through functions of the LIME[11] tuple-space middleware.

Capability Protocol. Given the networked system's interface signature, the behavior of the system's capabilities is specified as protocols over the system's actions defined in the interface signature. Such protocols need to be explicitly defined using some concurrent language, as part of the networked system's advertisements. Alternatively, the protocol specification may be learned in a systematic way based on the system's interfaces as investigated in the companion chapter on automata learning [18]. Different languages may be considered for such a specification from formal modeling to programming languages.

[10] As defined in the next section, *photoFile* and *photoComment* include photoID.

[11] http://lime.sourceforge.net

$\mathsf{Interface}_{photo\_sharing\_producer} = \{$
$< SOAP\text{-}RPCCall, Authenticate, < login >, \emptyset >,$
$< SOAP\text{-}RPCReceiveReply, Authenticate, \emptyset, < authenticationToken >>,$
$< SOAP\text{-}RPCCall, UploadPhoto, < photo >, \emptyset >$
$< SOAP\text{-}RPCReceiveReply, UploadPhoto, \emptyset, < acknowledgment >>$
$\}$
$\mathsf{Interface}_{photo\_sharing\_consumer} = \{$
$< SOAP\text{-}RPCCall, SearchPhotos, < photoMetadata >, \emptyset >,$
$< SOAP\text{-}RPCReceiveReply, SearchPhotos, \emptyset, < photoMetadataList >>,$
$< SOAP\text{-}RPCCall, DownloadPhoto, < photoID >, \emptyset >,$
$< SOAP\text{-}RPCReceiveReply, DownloadPhoto, \emptyset, < photoFile >>,$
$< SOAP\text{-}RPCCall, DownloadComment, < photoID >, \emptyset >,$
$< SOAP\text{-}RPCReceiveReply, DownloadComment, \emptyset, < photoComment >>,$
$< SOAP\text{-}RPCCall, CommentPhoto, < photoComment >, \emptyset >$
$< SOAP\text{-}RPCReceiveReply, CommentPhoto, \emptyset, < acknowledgment >>$
$\}$
$\mathsf{Interface}_{photo\_sharing\_server} = \{$
$< SOAP\text{-}RPCReceiveCall, Authenticate, < login >, \emptyset >,$
$< SOAP\text{-}RPCReply, Authenticate, \emptyset, < authenticationToken >>,$
$< SOAP\text{-}RPCReceiveCall, UploadPhoto, < photo >, \emptyset >,$
$< SOAP\text{-}RPCReply, UploadPhoto, \emptyset, < acknowledgment >>,$
$< SOAP\text{-}RPCReceiveCall, SearchPhotos, < photoMetadata >, \emptyset >,$
$< SOAP\text{-}RPCReply, SearchPhotos, \emptyset, < photoMetadataList >>,$
$< SOAP\text{-}RPCReceiveCall, DownloadPhoto, < photoID >, \emptyset >,$
$< SOAP\text{-}RPCReply, DownloadPhoto, \emptyset, < photoFile >>,$
$< SOAP\text{-}RPCReceiveCall, DownloadComment, < photoID >, \emptyset >,$
$< SOAP\text{-}RPCReply, DownloadComment, \emptyset, < photoComment >>,$
$< SOAP\text{-}RPCReceiveCall, CommentPhoto, < photoComment >, \emptyset >,$
$< SOAP\text{-}RPCReply, CommentPhoto, \emptyset, < acknowledgment >>$
$\}$

Fig. 17. Interface signature of infrastructure-based photo sharing

$\mathsf{Interface}_{photo\_sharing} = \{$
$< Out, PhotoMetadata, \emptyset, < photoMetadata >>,$
$< Out, PhotoFile, \emptyset, < photoFile >>,$
$< Rdg, PhotoMetadata, < photoMetadata >, < photoMetadataList >>,$
$< Rd, PhotoFile, < photoID >, < photoFile >>,$
$< Rd, PhotoComment, < photoID >, < photoComment >>,$
$< Out, PhotoComment, \emptyset, < photoComment >>,$
$< In, PhotoComment, < photoID >, < photoComment >>,$
$< Rd, PhotoComment, < photoID >, < photoComment >>$
$\}$

Fig. 18. Interface signature of Peer-to-Peer-based photo sharing

Formal languages are a prerequisite for automated reasoning about matching and mediator generation while well-established language from the Web service domain, such as BPEL[12], are easier for developer to deal with. Indeed, BPEL

[12] http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html

offers many advantages for the definition of processes, among which: (i) the specification of both data and control flows that allow identifying causally independent actions; (ii) the formal specification of BPEL in terms of process algebra that allows abstracting BPEL processes for automated reasoning about protocol matching [20]; and (iii) the rich tool sets coming along with BPEL, which in particular ease process definition by developers. However, same as for the interface signature definition, the language must be generalized to not be only specific to the Web service technology. Precisely, BPEL needs to be enriched so as to support interaction with networked systems using other interaction patterns and protocols than those classically associated with Web services, which can be addressed in a systematic way using the BPEL extension mechanism. Therefore, BPEL may be used by developers to specify the protocol implemented by the networked systems and automatically translated into FSP process algebra.

For illustration, Figure 3 gives the FSP-based specification of the protocols associated with a SOAP-based implementation of the infrastructure-based version of photo sharing application, while Figure 19 introduces the specification of a Lime-based implementation of the peer-to-peer version of the photo sharing application. The protocol executed by a Lime-based networked system allows for both production and consumption of photo files. On the other hand, there are different protocols for the producer, consumer and server for the SOAP-based implementation due to the distinctive roles imposed by the service implemented by the photo sharing server. Still, emergent connectors shall enable seamless

```
//Peer-to-Peer-based application specification
set Lime_PhotoSharing_Actions = {photoMetadata, photoFile, photoComment}
PhotoSharingPeer              = (req.photoMetadata →Consumer |prov.photoMetadata → Producer),
Producer                      = (prov.photoFile →PhotoSharingPeer),
Consumer                      = (req.photoFile →Consumer |req.photoComment →Consumer
                                |prov.photoComment →Consumer |req.photoFile →PhotoSharingPeer
                                |req.photoComment →PhotoSharingPeer
                                |prov.photoComment → PhotoSharingPeer |terminate →END).

//LIME middleware Specification
Lime_Reader(X =' tuple)       = (req.[X] →P1),
P1                            = (rd[X] →Lime_Reader |rdp[X] →Lime_Reader
                               | rdg[X] →Lime_Reader |in[X] →Lime_Reader
                               | inp[X] →Lime_Reader |ing[X] →Lime_Reader
                               | terminate → END).
Lime_Writer(X =' tuple)       = (prov.[X] →P2),
P2                            = (out[X] →Lime_Writer |outp[X] →Lime_Writer
                               | outg[X] →Lime_Writer |terminate →END).
Lime_glue(X =' tuple)         = (write[X] → P0 |outp[X] → P0 |outg[X] → P0
                               | terminate →END),
P0                            = (rd[X] →P0 |rdp[X] →P0 |rdg[X] →P0
                               | in[X] →Lime_glue |inp[X] →Lime_glue |ing[X] →Lime_glue).

const NumberOfPeers           = 2
||Lime_PhotoSharing           = ( [i : 1..NumberOfPeers]:PhotoSharingPeer
                                ||(forall [tuple:Lime_PhotoSharing_Actions] Lime_Writer(tuple))
                                ||(forall [tuple:Lime_PhotoSharing_Actions] Lime_Reader(tuple))
                                ||(forall [tuple:Lime_PhotoSharing_Actions] Lime_glue(tuple))).
```

Fig. 19. Peer-to-Peer-based photo sharing

interaction of the Lime-based photo sharing implementation with systems implementing capabilities of the infrastructure-based photo sharing.

4.2 Ontology for Mediation

Realizing emergent connectors primarily relies on reasoning about capability matching together with identifying matching observable actions among the actions performed by networked systems. Ontologies play a key role in identifying such matching and allow overcoming the inherent heterogeneity of pervasive networked systems. Indeed, "an ontology is a formal, explicit specification of a shared conceptualization" [49]. Such an ontology is then assumed to be shared widely. In addition, work on ontology alignment enables dealing with possible usage of distinct ontologies in the modeling of the different networked systems [17].

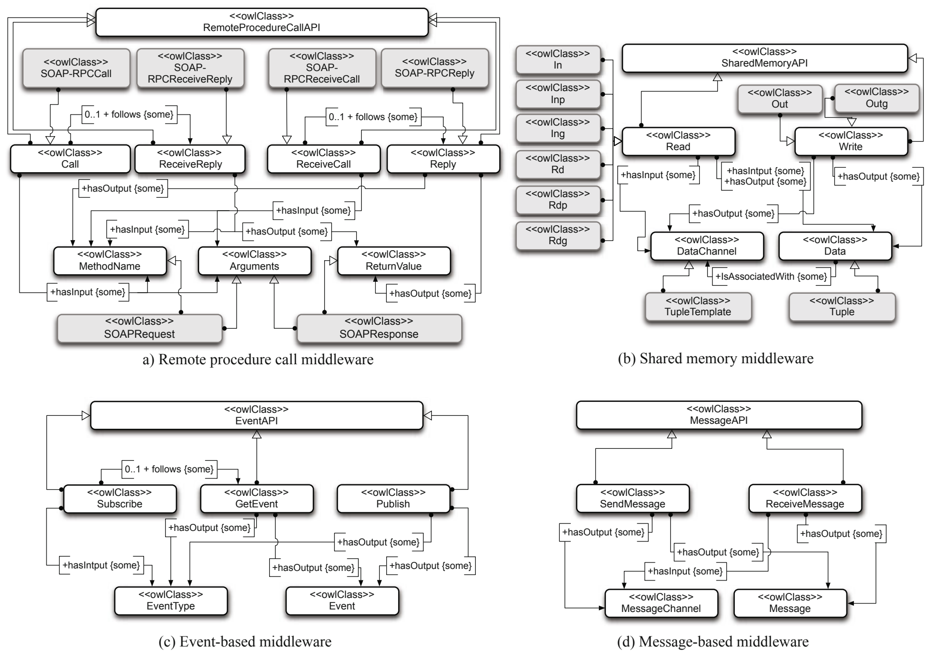
Different relations may be defined between ontology concepts. The *subsumption* relation (in general named *is-a*) is essential since it allows, besides equivalence, to match between concepts based on inclusion. Precisely: a concept C *is subsumed by* a concept D in a given ontology O, noted $C \sqsubseteq D$, if in every model of O the set denoted by C is a subset of the set denoted by D [2].

Towards enabling emergent connectors, we introduce a middleware ontology that forms the basis of middleware protocol mediation. In addition, domain-specific application ontologies characterizing application actions serve defining both control- and data-centric concepts.

Middleware Ontology. As discussed in Section 2.1, state-of-the-art middleware may be categorized according to four *middleware types* regarding provided communication and coordination services [50]: *remote procedure call, shared memory, event-based* and *message-based.* As depicted in Figure 20 and more specifically with concepts defined in white boxes, the proposed middleware ontology is structured around these four categories, which serve as reference enabling to align functions of different middleware solutions. Indeed, the reference middleware ontology can be refined into concepts associated with functions of a specific middleware. This is illustrated in the figure by the grayed boxes that define concepts of the Lime and SOAP-based middleware solutions that we specifically consider in our photo sharing scenario. In addition to the *is-a* relation that is denoted by a white arrow, the middleware ontology introduces a number of customized relations between concepts: *hasOutput* (resp. *hasInput*) to characterize output (resp. input) parameters. We also use relations from best practices in ontology design[13] as illustrated by the *follows* relation that serves defining sequence patterns.

The ontology is given as a set of UML diagrams. In Figure 20.a), the ontology concepts associated with RPC-based middleware include the *Call* function parameterized by the method name and arguments, which must must be followed by the *ReceiveReply* function to receive the result of the call. On the server side, the *ReceiveCall* function to catch an invocation is followed by the execution of the *Reply* function to return the result. The ontologies of functions for

[13] http://ontologydesignpatterns.org



a) Remote procedure call middleware (b) Shared memory middleware

(c) Event-based middleware (d) Message-based middleware

Fig. 20. Middleware ontology

shared memory and message-based middleware are rather straightforward. In the former, the shared memory is accessed through *Read*/ *Write* functions parameterized by the associated data and corresponding channel (see Figure 20.b). In the latter, messages are exchanged using the *SendMessage* and *ReceiveMessage* functions parameterized by the actual message and related channel (see Figure 20.d). Regarding event-based middleware, events are published using the *Publish* function parameterized by the specific event; while they are consumed through the *GetEvent* function after registering for the specific event type using the *Subscribe* function (see Figure 20.c).

The proposed ontology serves aligning the functions of middleware of the same type through mapping onto the reference functions, which is illustrated for the specific cases of SOAP-based and Lime middleware. Heterogeneity in the underlying implementation may then be overcome using transparent middleware interoperability solutions (see Section 3.3).

A further challenge for emergent connectors in pervasive networking environments is to enable mediation among different middleware types. To enable such mediation, we introduce a further abstraction allowing cross-type alignment of middleware functions. More specifically, according to their semantics, middleware functions may be aligned based on whether they produce or consume an action in the network. We hence define the mapping of middleware functions onto abstract *input* and *output* (denoted by an overbar) actions, which are parameterized by the application action a and associated input I and output O.

The alignment of (possibly sequence of) middleware functions as abstract input and output actions is summarized in Figure 21. The alignment defined for shared memory and message-based middleware functions is rather direct: the *Write* and *SendMessage* functions are mapped onto an output action; while the *Read* and *ReceiveMessage* translate into an input action. Note that *Read* is possibly parameterized with *I* if the value to be read shall match some constraints, as, e.g., enabled by tuple space middleware. The alignment for the event-based middleware functions is straightforward for *Publish*: publication of an event maps onto an output action. The dual input action is performed by the *GetEvent* function, which is preceded by at least one invocation of *Subscribe* on the given event[14]. The semantics of RPC functions follows from the fact that it is the server that produces an application action, although this production is called upon by the client. Then, the output action is defined by the execution of *ReceiveCall* followed by *Reply*, while the dual input action is defined by the *Invoke* function.

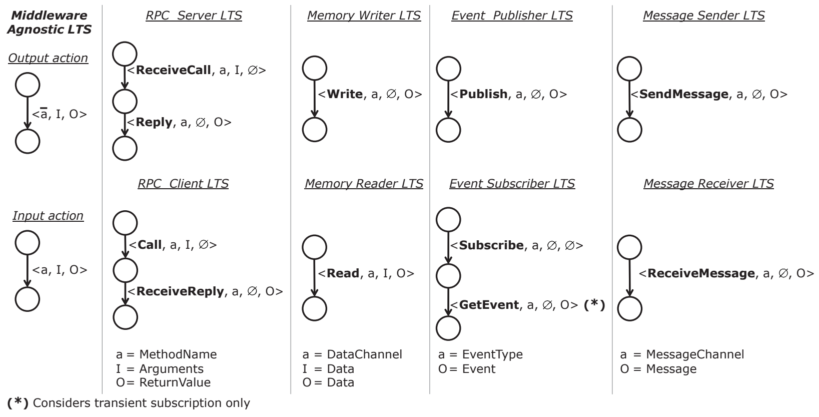


Fig. 21. Middleware alignment

The given alignments abstract protocols associated with the realization of capabilities as *middleware-agnostic processes*. As a result, protocols may be matched based purely on their application-specific features. In more detail, middleware-specific functions are abstracted as middleware functions from the reference ontology, which are then translated into input and output actions through the defined alignment. This is illustrated in Figure 22, which gives the FSP-based protocol associated with the peer-to-peer photo sharing implementation, after abstracting middleware-specific functions into reference functions (see Figure 23) and further aligning onto middleware-agnostic input and output actions. Thanks to the alignment of middleware functions, processes may be

[14] Note that for the sake of conciseness, the figure depicts only the case where a *Subscribe* is followed by a single *GetEvent*.

```
Reader(X =' data)   = (req.[X] →P1),
P1                  = (read[X] →Reader |terminate → END).
Writer(X =' data)   = (prov.[X] →P2),
P2                  = (write[X] →Writer |terminate →END).
SM_glue(X =' data)  = (write[X] → P3 |terminate →END),
P3                  = (read[X] →SM_glue).
```

Fig. 22. Shared-memory middleware type specification

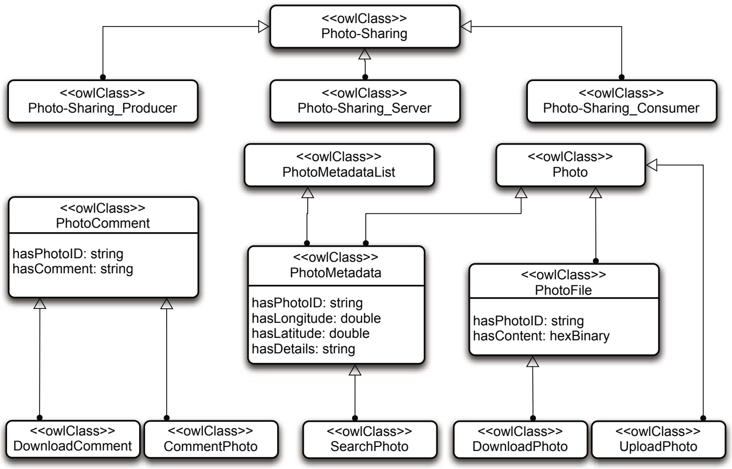
```
Reader(X =' data)   = (input[X] →Reader|terminate → END).
Writer(X =' data)   = (output[X] →Writer|terminate → END).
SM_glue(X =' data)  = (output[X] → P |terminate →END),
P                   = (input[X] →SM_glue).
```

Fig. 23. Middleware-agnostic peer-to-peer photo sharing

matched against the realization of matching application-specific actions whose semantics is given by the associated ontology.

Application-Specific Ontology. The *subsumption* relation of ontologies serves matching application-specific capabilities and actions against each other. Basically, and as detailed in the next section, a required capability/action matches a provided one if the former is subsumed by the latter.

For illustration, Figure 24 gives an excerpt of the domain-specific ontology associated with our photo sharing scenario, which shows the subsumptions holding among the various concepts defining the interfaces of the networked systems implementing the scenario.



<<owlClass>> Photo-Sharing
<<owlClass>> Photo-Sharing_Producer
<<owlClass>> Photo-Sharing_Server
<<owlClass>> Photo-Sharing_Consumer
<<owlClass>> PhotoMetadataList
<<owlClass>> Photo
<<owlClass>> PhotoComment
hasPhotoID: string
hasComment: string
<<owlClass>> PhotoMetadata
hasPhotoID: string
hasLongitude: double
hasLatitude: double
hasDetails: string
<<owlClass>> PhotoFile
hasPhotoID: string
hasContent: hexBinary
<<owlClass>> DownloadComment
<<owlClass>> CommentPhoto
<<owlClass>> SearchPhoto
<<owlClass>> DownloadPhoto
<<owlClass>> UploadPhoto

Fig. 24. Photo sharing ontology

Note that the application-specific ontology not only describes the semantics and relationships related to data but also to the functionalities and roles of the networked systems, such as *Photo-Sharing_Producer*, *Photo-Sharing_Consumer*, and *Photo-Sharing_Server*. It also defines the semantics of the operations performed on data, such as *UploadPhoto*, *DownloadPhoto*, and *SearchPhoto*. Furthermore, it relates data to operations: data subsumes the operations performed on them. The rationale behind this statement is that by having access to data, any operation could be performed on it. For example *PhotoFile* subsumes *DownloadPhoto* since by providing access to a photo file, one can download it.

Finally, subsumption is not the panacea to reason about semantic relationships between concepts and many other relations such as sequence [16] or part-whole[15] should be specified. We believe that best practices of ontology design and ontology engineering[16] and the use of ontology design patterns[17] may prove very beneficial to automatically discover and reuse semantic relations between concepts.

4.3 Emergent Connectors

Given the models characterizing networked systems that are introduced in Section 4.1 and related ontology definition, emergent connectors are enabled through matching and mapping functions defined over the actions of networked systems. Precisely, if two networked systems implement matching capabilities, then they may possibly coordinate towards the realization of the capability. This is achieved by mapping the respective actions of the systems according to their ontology-based semantics, and then synthesizing the mediator that adapts accordingly the interaction protocols executed by the networked systems.

Capability Matching. The first step in identifying the possible matching of two networked systems is to assess whether they respectively provide and require a matching capability. Precisely, and following the definition of semantic matching of capabilities [41], we say that capability $C_R =< Req, \mathcal{C}_R, I_R, O_R >$ semantically matches with capability $C_P =< Prov, \mathcal{C}_P, I_P, O_P >$, noted $C_R \hookrightarrow C_P$, *iff* in the given ontology:

- $\mathcal{C}_R \sqsubseteq \mathcal{C}_P$,
- $I_P \sqsubseteq I_R$ (which is a shorthand notation for subsumption between sets of ontology concepts), and
- $O_R \sqsubseteq O_P$.

Note that a capability C_R of type *Req produces* the inputs I_R and *consumes* the corresponding outputs O_R. In a dual manner, a capability C_P of type *Prov consumes* the inputs I_P and *produces* the corresponding outputs O_P.

In addition, since the capability is related to semantic concepts, we make a similar assumption to that made in the Semantic Web [41], i.e., by specifying $\mathcal{C}_P$

[15] `http://www.w3.org/2001/sw/BestPractices/OEP/SimplePartWhole/index.html`
[16] `http://www.w3.org/2001/sw/BestPractices/OEP/`
[17] `http://ontologydesignpatterns.org`

as the functional concept, the provider commits to offering all the functionalities subsumed by $\mathcal{C}_P$ and output consistent with every concept subsumed by O_P. If this is not the case, then the functionality/output should be restricted to those verifying the above assumption. Similarly, the requester commits to provide any input consistent with the classes that I_R subsumes. However, if the input/output are related to syntactic (XML-based) types and not to semantic concepts, it becomes important to verify the Liskov Substitution Principle (LSP) [32] in the following way:

– $\mathcal{C}_P$ `subtypeOf` $\mathcal{C}_R$, which corresponds to the LSP co-variance rule;
– I_R `subtypeOf` I_P, which corresponds to the LSP contra-variance rule for the outputs; and
– O_P `subtypeOf` O_R, which corresponds to the LSP co-variance rule.

That being the case, if the semantic concept is automatically extracted or learned from the syntactic description, then it should be restricted to the most specific concept. Moreover, since there is a close relation between the semantic concepts and the related syntactic objects, it is required to have specifications or methods enabling transformations between the different concepts and types.

In the case where one capability is required and provided (i.e., of type *Req_Prov*) by a networked system and the other capability is required (resp. provided) by the other networked system, the same condition as above applies considering that the *Req_Prov* capability is considered as being provided and required. For instance, given (1) and (2) below, we have (3):

$$\begin{aligned} &\mathit{PhotoSharingConsumer} = <\mathit{Req}, \mathit{Photo-Sharing\_Consumer}, <\mathit{PhotoComment}>, <\mathit{Photo}>> && (1)\\ &\mathit{PhotoSharing} = <\mathit{Req\_Prov}, \mathit{Photo-Sharing}, <\mathit{Photo}> \vee <\mathit{PhotoComment}>,\\ &\qquad <\mathit{Photo}, \mathit{PhotoComment}>> && (2)\\ &\mathit{PhotoSharingConsumer} \hookrightarrow \mathit{PhotoSharing} && (3) \end{aligned}$$

Given capability matching, the emergent connector between the matching networked systems should mediate possible behavioral mismatches in their respective middleware-agnostic interaction protocols. Towards that goal, we build on basic mediation patterns.

Mediation Patterns. Possible behavioral mismatches for input actions need to be solved so as to ensure that any input action is synchronized with an output action of the matching networked system with respect to the realization of the capability of interest. On the other hand, the absence of consumption of an output action does not affect the behavior of the networked system as long as deadlock is prevented by the emergent connector at runtime. Still, synthesis of a protocol mediator is known as a computationally hard problem for finite state systems in general [11] and thus requires heuristics to make the problem tractable. Towards that goal, we focus on enabling basic mediation patterns [45] as introduced in the literature for, e.g., Semantic Web Services [48]. We then account for basic mediation patterns as follows:

- **Ordering mismatch:** This concerns the re-ordering of actions so that networked systems may indeed coordinate. In the case of BPEL specification, causally independent actions may be identified through data-flow analysis, hence enabling to introduce concurrency among actions and thus supporting acceptable re-ordering.
- **Extra output action (or missing input action):** As discussed above, extra output actions are simply discarded from the standpoint of behavioral matching. Obviously, the associated concrete mediator should handle any extra synchronous output action to avoid deadlock.
- **Extra input action (or missing output action):** Any input action needs to be mapped to an output action of the matching networked system. However, in this case, there is no such output action that directly maps to the input action. In a first step, we do not handle these mismatches as they would significantly increase the complexity of protocol adaptation.
- **Splitting of actions:** Splitting actions relate to having an action of one system realized by a number of actions of the other. Then, an input action may be split into a number of output actions of the matching networked system if such a relation holds from the domain-specific ontology giving the semantics of actions. On the other hand, we do not deal with the splitting of output actions, which is an area for future work given the complexity it introduces.
- **Merging of actions:** The merging of actions is the dual of splitting from the standpoint of the matching networked system. Then, we only handle the merging of output actions.

Interface Mapping. Following the above, interface mapping serves identifying mapping among the actions of the interaction protocols run by the networked systems that should coordinate towards the realization of a given capability.

Let two networked systems that respectively implement the matching capabilities $\mathcal{C}_1$ and $\mathcal{C}_2$. Let further $\mathcal{I}_{\mathcal{C}_1}$ (resp. $\mathcal{I}_{\mathcal{C}_2}$) be the set of middleware-agnostic actions executed by the protocol realizing $\mathcal{C}_1$ (resp. $\mathcal{C}_2$); $\mathcal{I}_{\mathcal{C}_1}$ and $\mathcal{I}_{\mathcal{C}_2}$ are then subsets of the actions defined in the networked systems' interfaces, which are further made middleware-agnostic according to the alignment defined in Section 4.2. We introduce the function $Map_I(\mathcal{I}_{\mathcal{C}_1}, \mathcal{I}_{\mathcal{C}_2})$ which identifies the set of all possible mappings of all the input actions of $I_{\mathcal{C}_1}$ (resp. $I_{\mathcal{C}_2}$) with actions of $\mathcal{I}_{\mathcal{C}_2}$ (resp. $\mathcal{I}_{\mathcal{C}_1}$), according to the semantics of actions. Formally:

$$\begin{aligned} Map_I(\mathcal{I}_{\mathcal{C}_1}, \mathcal{I}_{\mathcal{C}_2}) \quad &= \bigcup_{<a,I,O>\in\mathcal{I}_{\mathcal{C}_1}} \{< a, I, O > \mapsto map(< a, I, O >, \mathcal{I}_{\mathcal{C}_2})\} \bigcup \\ &\bigcup_{<a',I',O'>\in\mathcal{I}_{\mathcal{C}_2}} \{< a', I', O' > \mapsto map(< a', I', O' >, \mathcal{I}_{\mathcal{C}_1})\} \end{aligned}$$

where:

$$\begin{aligned} map(< a, I_a, O_a >, \mathcal{I}) = \{ &<< \overline{b}_i, I_i, O_i > \in \mathcal{I} >_{i=1..n} \mid \\ &\quad a \sqsubseteq \cup_i\{b_i\} \\ &\wedge I_{i\leq n} \sqsubseteq (\cup_{j<i}\{O_j\}) \cup \{I_a\} \\ &\wedge O_a \sqsubseteq (\cup_{j<i}\{O_j\}) \cup \{I_a\} \\ &\} \end{aligned}$$

and

$$\forall seq_1 \in map(< a, I_a, O_a >, \mathcal{I}),\ \nexists seq_2 \in map(< a, I_a, O_a >, \mathcal{I}) | seq_2 \prec seq_1$$

where $\prec$ denotes the inclusion of sequences. In the above definition, the ordering of actions given by the sequence follows from the sequencing of actions in the protocol realizing the capability. The definition is further given in the absence of concurrent actions to simplify the notations, while the generalization to concurrent actions is rather direct.

As an illustration, we give below the interface mapping between the *PhotoSharingConsumer* and *PhotoSharing* capabilities. All the input actions of *PhotoSharingConsumer* have a corresponding output action in *PhotoSharing*. On the other hand, the input actions of *PhotoSharing* associated with the production of photos do not have matching output actions in *PhotoSharingConsumer*. As a result, we support the adaptation of protocols for interaction between *PhotoSharingConsumer* and *PhotoSharing* regarding the consumption of photos by the former only, as further discussed in the next section.

$$
\begin{array}{l}
\text{Map}(\text{Interface'}_{photo\_sharing\_consumer}, \text{Interface}'_{photo\_sharing}) = \{ \\
\quad < SearchPhotos, photoMetadata, photoMetadataList > \\
\quad\quad \mapsto \{<< \overline{PhotoMetadata}, \emptyset, photoMetadata >>\}, \\
\quad < DownloadPhoto, photoID, photoFile > \\
\quad\quad \mapsto \{<< \overline{PhotoFile}, \emptyset, photoFile >>\}, \\
\quad < CommentPhoto, photoComment, acknowledgment > \\
\quad\quad \mapsto \{<< \overline{PhotoComment}, \emptyset, photoComment >>\}, \\
\quad < DownloadComment, photoID, photoComment > \\
\quad\quad \mapsto \{<< \overline{PhotoComment}, \emptyset, photoComment >>\}, \\
\quad < PhotoComment, photoID, photoComment > \mapsto \emptyset, \\
\quad < PhotoMetadata, photoMetadata, photoMetadataList > \mapsto \emptyset, \\
\quad < PhotoFile, photoID, photoFile > \mapsto \emptyset \\
\}
\end{array}
$$

Mediator Synthesis. Given interface mappings returned by Map_I, we need to identify whether the protocols associated with the matching capabilities may indeed coordinate, i.e., the concurrent execution of the two protocols successfully terminates. However, in a first step , we assume that it exists a single mapping for each input action. Formally, let:

$$\mathcal{I}'_1 = \{\alpha_i = \langle a_i, I_{a_i}, O_{a_i} \rangle\}_{i=1..n} \cup \left\{\overline{\beta_j} = \left\langle \overline{b_j}, I_{b_j}, O_{b_j} \right\rangle\right\}_{j=1..m}$$

be the abstract interface associated with required capability $\mathcal{C}_1$, and:

$$\mathcal{I}'_2 = \left\{\alpha'_{i'} = \left\langle a'_{i'}, I_{a_{i'}}, O_{a_{i'}} \right\rangle\right\}_{i'=1..n'} \cup \left\{\overline{\beta'_{j'}} = \left\langle \overline{b'_{j'}}, I_{b'_{j'}}, O_{b'_{j'}} \right\rangle\right\}_{j'=1..m'}$$

be the abstract interface associated with provided capability $\mathcal{C}_2$.

From $Map_I(\mathcal{I}'_1, \mathcal{I}'_2)$, we have:

$$\forall \alpha_{i=1..n} \in \mathcal{I}'_1 : \alpha_i \mapsto \left\langle \overline{\beta'_1}, ..., \overline{\beta'_n} \right\rangle \mid \beta'_j \in \mathcal{I}'_2$$

We then define the processes $\mathsf{M}_{\alpha_{i=1..n}}$ that deal with the splitting/merging of $\mathcal{C}_1$ actions by allowing the synchronization of each input action $\alpha_{i=1..n}$ with its corresponding output actions:

$$\mathsf{M}_{\alpha_{i=1..n}} = \beta'_1 \rightarrow ... \rightarrow \beta'_n \rightarrow \overline{\alpha_i} \rightarrow \mathsf{M}_{\alpha_{i=1..n}}$$

We further define the processes $\mathsf{M}_{\beta'_{j'=1..k'}}$ for any extra output action $\beta'_{j'} \in \mathcal{I}'_2$ that is not required by any input action $\alpha_{i=1..n} \in \mathcal{I}'_1$, as follows:

$$\mathsf{M}_{\beta'_{j'=1..k'}} = \beta'_{j'=1..k'} \rightarrow \mathsf{M}_{\beta'_{j'=1..k'}}$$

We define similarly $\mathsf{M}_{\alpha'_{i'=1..n'}}$ and $\mathsf{M}_{\beta_{j=1..k}}$ for $\mathcal{C}_2$.

A process P_1 associated with capability $\mathcal{C}_1$ *behaviorally matches* a process P_2 associated with capability $\mathcal{C}_2$ under $Map(\mathcal{I}'_1, \mathcal{I}'_2)$, noted $P_1 \hookrightarrow_{\mathcal{P}} P_2$, iff

$$P_1 \underset{i=1..n}{||} \mathsf{M}_{\alpha_{i=1..n}} \underset{j'=1..k'}{||} \mathsf{M}_{\beta'_{j'=1..k'}} \leq P_2 \underset{i'=1..n'}{||} \mathsf{M}_{\alpha'_{i'=1..n'}} \underset{j=1..k}{||} \mathsf{M}_{\beta_{j=1..k}}$$

where $\leq$ refers to trace refinement as defined in [24] and guarantees that *mediated* P_1 can safely communicate with *mediated* P_2.

Applying the above definition, we can check that:

$$P_{photo\_sharing\_consumer} \hookrightarrow_{\mathcal{P}} P_{photo\_sharing}$$

Consequently, the *emergent connector mediator* is defined as follows:

$$||\mathsf{Emergent\_Connector\_Mediator} =$$
$$\left(\underset{i=1..n}{||} \mathsf{M}_{\alpha_{i=1..n}} \right) || \left(\underset{j'=1..k'}{||} \mathsf{M}_{\beta'_{j'=1..k'}} \right) || \left(\underset{i'=1..n'}{||} \mathsf{M}_{\alpha'_{i'=1..n'}} \right) || \left(\underset{j=1..k}{||} \mathsf{M}_{\beta_{j=1..k}} \right)$$

4.4 From Abstract to Concrete Emergent Connectors

Once the model of the emergent connector has been synthesized, it needs to be transformed into a concrete software artifact. The concretization is threefold:

1. Parsing the network messages in order to generate the corresponding actions; this parsing is performed by a *Listener* specific to each middleware implementation (see Figure 25A).
2. Generating the code corresponding to the mediator (see Figure 25B).
3. Composing the abstract actions in order to generate the corresponding network message; this is the role of an *Actuator* specific to each middleware implementation (see Figure 25C).

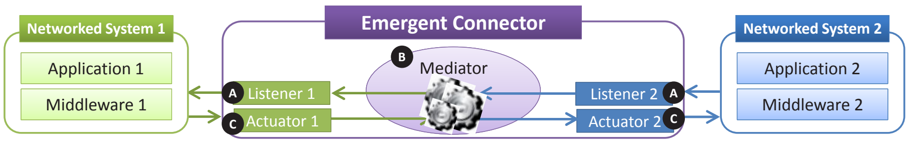


Fig. 25. Concretizing the mediator

Towards the above, we adopt results in the area of synthesis of concrete middleware protocols. Indeed, these last years two main approaches, z2z [10] and Starlink [8], have emerged to synthesize middleware, which acts as gateways to translate one protocol to another. More precisely, these approaches have instantiated the direct bridge concepts, as it provides a high degree of expressiveness and does not require modifications to existing applications. Both z2z and Starlink are based on similar concepts (see Figure 26a,b): they provide an optimized run-time system, and facilities for describing network protocol behaviors, message structures, and translation logics. Such facilities come from the fact that they rely on a high-level definition language that hides low level network details and highlights only key properties of protocols. Hence, to get a generated gateway between two heterogeneous protocols, developers must write specifications consisting of: (i) a protocol specification, describing how the protocols interact with the network, (ii) a message specification, describing the structure of message requests and responses, and (iii) a translation specification, describing how to translate messages among protocols (See Figure 26, ❶,❷). These specifications enable to generate software components such as *listeners*, *actuators* and *mediators* that are plugged into a runtime system to form, from a formal point of view, a *direct bridging* connector (as introduced in Section 3.1). *Listeners*, and *actuators* enable respectively to extract required informations relevant to the interacting parties, and to generate extracted informations in an adequate format according to protocols being used. The *mediator* applies the required translation logic to resolve mismatches between protocols.

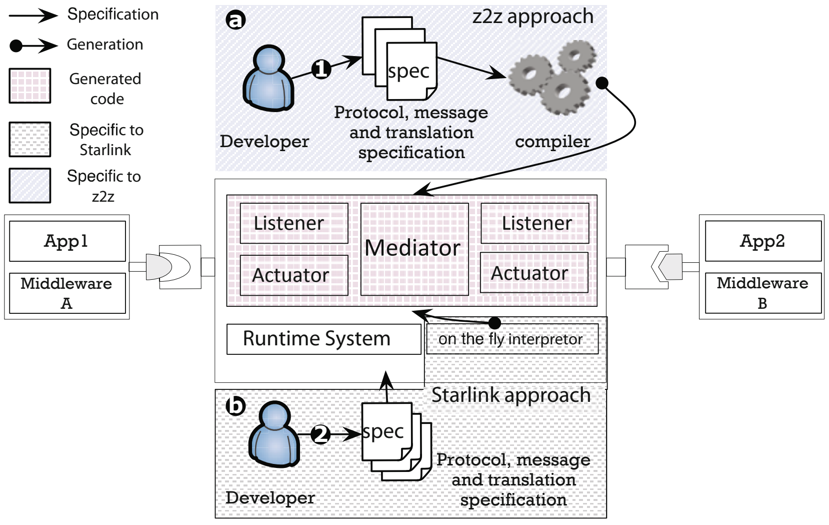


Fig. 26. z2z and Starlink approaches to synthesize middleware

Although z2z and Starlink are closed together in their design, they differ strongly in the way code plugged into the runtime is generated. With z2z, generated gateways are statically built. Hence, once such gateways are deployed in one environment, it is not anymore possible to alter afterwards the translation being processed. Consequently, in environments where systems are composed dynamically, interoperability can not be guaranteed. In general, z2z targets environments where gateways need to be embedded in resource constraint devices with performances in mind. Specifications in z2z are expressed in a C-like language and are compiled at design time. The z2z compiler relies on advanced compilation strategies to perform static verifications at the specification level and to produce highly optimized native code dedicated to the translation between two specific protocols. On the contrary, Starlink is designed with both dynamicity and genericity in mind. Specifications in Starlink are processed dynamically at runtime, and the code plugged into the runtime is done on the fly according to protocols currently used in the environment (See Figure 26b). To this end, compared to z2z, the Starlink runtime embeds both generic parsers and composers that are customized dynamically according to the specifications being used. It is important to note that in Starlink, specifications are interpreted and not compiled as in z2z. Hence, it has a potential impact on performance.

4.5 Related Work

Protocol interoperability has been the focus of significant research since the early days of networking. This has initially led to the study of systematic approaches to protocol conversion (i.e., synthesizing a mediator that adapts the two interacting protocols that need to interoperate) based on formal methods as surveyed in [11]. Existing approaches may in particular be classified into two categories depending on whether: (i) they are bottom-up, heuristic-based, or (ii) top-down, algorithmic-based. In the former case, the conversion system derives from some given protocol, which may either be inferred from the semantic correspondence between the messages of the interacting protocols [30] or correspond to the reference protocol associated with the service to be realized through protocol interaction [39]. In the latter case, protocol conversion is considered as finding the quotient between the two interacting protocols. Then, if protocols are specified as finite-state systems, an algorithm computing the quotient is possible but the problem is computationally hard since it requires an exhaustive search of possibilities [11]. Then, the advantage of the bottom-up approach is its efficiency but at the expense of: (i) requiring the message mapping or reference protocol to be given and further (ii) not identifying a converter in all cases. On the other hand, the bottom-up approach will always compute a converter if it exists given the knowledge about the semantics of messages, but at the expense of significant complexity. This has led to the further development of formal approaches to protocol conversion so as to improve the performance of proposed algorithms [29]. Our work extensively builds on these formal foundations, adopting a bottom-up approach in the form of interface mapping. However, unlike the work of [30], our interface mapping is systematically inferred, thanks to the use of ontologies.

In addition, while the proposed formal approaches pave the way for rigorous reasoning about protocol compatibility and conversion, they are mostly theoretical, dealing with simple messages (e.g., absence of parameters).

More practical treatment of protocol conversion is addressed in [52], which focuses on the adaptation of component protocols for object-oriented systems. The solution is top-down in that the synthesis of the mediator requires the mapping of messages to be given. By further concentrating on practical application, the authors have primarily targeted an efficient algorithm for protocol conversion, leading to a number of constraining assumptions such as synchronous communication. In general, the approach is quite restrictive in the mediation patterns that it supports by not buffering messages and thus preventing the handling of the merging/splitting or re-ordering of messages in general. Then, while our solution relates to this specific proposal, it is more general by dealing with more complex mediation patterns and further inferring message mapping from the ontology-based specification of interfaces. Our solution further defines protocol compatibility by in particular requiring that any input action (message reception) has a corresponding (set of) output action(s), while the definition of [52] requires the reverse. Our approach then enforces networked systems to coordinate so as to update their states as needed, based on input from the environment.

More recently, with the emergence of Web services and advocated universal interoperability, the research community has been investigating how to actually support service substitution so as to enable interoperability with different implementations (e.g., due to evolution or provision by different vendors) of a service. While early work has focused on semi-automated, design-time approaches [37,42], latest work concentrates on automated, run-time solutions [12]. Our work closely relates to the latest effort, sharing the exploitation of ontology to reason about interface mapping and the further synthesis of protocol converter behaviors according to such mapping, using model checking [12]. However, our work goes one step further by not being tight to the specific Web service domain but instead considering highly heterogeneous pervasive environments where networked systems may build upon diverse middleware technologies and hence protocols.

Our work also closely relates to significant effort from the semantic Web service domain and in particular the WSMO (Web Service Modeling Ontology) initiative that defines mediation as a first class entity for Web service modeling towards supporting service composition. The resulting Web service mediation architecture highlights the various mediations levels that are required for systems to interoperate in a highly open network [48]: data level, functional level, and process level. This has in particular led to elicit base patterns for process mediation together with supporting algorithms [14,51]. However, as for the above-mentioned work on Web service adaptation, mediation is focused on the upper application layer, ignoring possible mismatches in the lower protocol layers. In other words, work from the Web service arena so far concentrates on interoperability among networked systems from the same technology domain. However, pervasive networks will increasingly be populated by highly heterogeneous systems, spanning, e.g., from systems for sensing/actuating to enterprise information systems. As a result, systems run disparate middleware protocols that need to be reconciled on the fly.

The issue of middleware interoperability has deserved a great deal of attention since the emergence of middleware. Solutions were initially dealing with diverging implementations of the same middleware specification and then evolved to address interoperability among different middleware solutions, acknowledging the diversity of systems populating the increasingly complex distributed systems of systems. As already discussed, one-to-one bridging was among the early approaches [40] and then evolved into more generic solutions such as Enterprise Service Bus [13], interoperability platforms [21] and transparent interoperability approaches [9,36]. Our work takes inspiration from the latest transparent interoperability approach, which is itself based on early protocol conversion approaches. Indeed, protocol conversion appears the most flexible approach as it does not constrain the behavior of networked systems. Then, our overall contribution comes from the comprehensive handling of protocol conversion, from the application down to the middleware layers, which have so far been tackled in isolation. In addition, existing work on middleware-layer protocol conversion focuses on interoperability between middleware solutions implementing the same interaction paradigm. On the other hand, our approach allows for interoperability among networked systems based upon heterogeneous middleware paradigms, which is crucial for the increasingly heterogeneous pervasive networking environment.

5 Conclusion

The need to deal with the existence of different protocols that perform the same function is not new and has been the focus of tremendous work since the 80s, leading to the study of protocol mediation from both theoretical and practical perspectives. However, while this could be initially considered as a transitory state of affairs, the increasing pervasiveness of networking technologies together with the continuous evolution of information and communication technologies make protocol interoperability a continuous research challenge. As a matter of fact, networked systems now need to compose on the fly while overcoming protocol mismatches from the application down to the middleware layer. Towards that goal, this paper has discussed the foundations of *emergent connectors*, which adapt the protocols run by networked systems that implement a matching functionality but possibly mismatch from the standpoint of associated application protocol and even middleware technology used for interactions. Enabling emergent connectors specifically lies in the appropriate modeling of the networked systems' high-level functionalities and related protocols, for which we exploit ontologies so as to enable unambiguous specification. Compared to related work that deals with either automated protocol conversion/mediation or middleware interoperability, our contribution lies in comprehensively dealing with both the application and middleware layers. In addition, through the alignment of middleware concepts, we are able to deal with interoperability between networked systems relying on heterogeneous middleware paradigms.

While this paper has surveyed the overall model-based approach enabling emergent connectors, it comes along with concrete enablers to be deployed in the network for actual enactment of the connectors [4], as studied in companion chapter on the Connect architecture [22]. Enablers include *universal discovery*, which in particular implements the matching and mapping relations discussed in this paper, so as to enable networked systems to meet and compose on the fly. However, it should be acknowledged that most legacy systems do not advertise interfaces like the ones needed by emergent connectors but instead advertise simple interface signatures, as common with today's middleware. This leads the Connect project to investigate *learning enablers* so as to enable automated learning of interaction protocols [5,25] as well as inference of capabilities from interface signatures. Furthermore, while universal discovery enables networked systems to compose abstractly through the proposed model-based approach to emergent connection, concrete connectors need to be instantiated, which concretize the proposed model-based protocol conversion according to actual middleware protocols and application actions. Concretization of mediation processes is in particular investigated based on the exploitation of domain-specific languages as defined in Section 4.4. Preliminary prototypes of the Connect enablers are being implemented and will be shortly released on the Connect Web site [28].

Acknoledgment. This work is partially supported by the FP7 ICT FET IP Project CONNECT. The authors would also like to thank their colleagues Nikolaos Georgantas, Rachid Saadi, Romina Spalazzese and Paola Inverardi for useful discussion about this work.

References

1. Allen, R., Garlan, D.: A formal basis for architectural connection. ACM Trans. Softw. Eng. Methodol. 6(3) (1997)
2. Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P.F.: The Description Logic Handbook. Cambridge University Press, Cambridge (2003)
3. Ben Mokhtar, S., Preuveneers, D., Georgantas, N., Issarny, V., Berbers, Y.: EASY: Efficient semantic service discovery in pervasive computing environments with QoS and context support. Journal of Systems and Software 81(5) (2008)
4. Bennaceur, A., Blair, G.S., Chauvel, F., Huang, G., Georgantas, N., Grace, P., Howar, F., Inverardi, P., Issarny, V., Paolucci, M., Pathak, A., Spalazzese, R., Steffen, B., Souville, B.: Towards an architecture for runtime interoperability. In: Margaria, T., Steffen, B. (eds.) ISoLA 2010. LNCS, vol. 6416, pp. 206–220. Springer, Heidelberg (2010)
5. Bertolino, A., Inverardi, P., Pelliccione, P., Tivoli, M.: Automatic synthesis of behavior protocols for composable web-services. In: Proceedings of ESEC/SIGSOFT FSE (2009)
6. Blair, G., Paolucci, M., Grace, P., Georgantas, N.: Interoperability in complex distributed systems. In: Bernardo, M., Issarny, V. (eds.) SFM 2011. LNCS, vol. 6659, pp. 350–392. Springer, Heidelberg (2011)
7. Bromberg, Y.D.: Solutions to middleware heterogeneity in open networked environment. Ph.D. thesis, Université de Versailles Saint-Quentin-en-Yvelynes (2006)

8. Bromberg, Y.D., Grace, P., Réveillère, L.: Starlink: runtime interoperability between heterogeneous middleware protocols. In: Proceedings of ICDCS 2011. IEEE Computer Society, Los Alamitos (2011)
9. Bromberg, Y.D., Issarny, V.: INDISS: Interoperable discovery system for networked services. In: Alonso, G. (ed.) Middleware 2005. LNCS, vol. 3790, pp. 164–183. Springer, Heidelberg (2005)
10. Bromberg, Y.D., Réveillère, L., Lawall, J.L., Muller, G.: Automatic generation of network protocol gateways. In: Bacon, J.M., Cooper, B.F. (eds.) Middleware 2009. LNCS, vol. 5896, pp. 21–41. Springer, Heidelberg (2009)
11. Calvert, K.L., Lam, S.S.: Formal methods for protocol conversion. IEEE Journal on Selected Areas in Communications 8(1) (1990)
12. Cavallaro, L., Nitto, E.D., Pradella, M.: An automatic approach to enable replacement of conversational services. In: Baresi, L., Chi, C.-H., Suzuki, J. (eds.) ICSOC-ServiceWave 2009. LNCS, vol. 5900, pp. 159–174. Springer, Heidelberg (2009)
13. Chappell, D.A.: Enterprise Service Bus. O'Reilly, Sebastopol (2004)
14. Cimpian, E., Mocan, A.: WSMX process mediation based on choreographies. In: Bussler, C.J., Haller, A. (eds.) BPM 2005. LNCS, vol. 3812, pp. 130–143. Springer, Heidelberg (2006)
15. Denaro, G., Pezzè, M., Tosi, D.: Ensuring interoperable service-oriented systems through engineered self-healing. In: Proceedings of ESEC/SIGSOFT FSE (2009)
16. Drummond, N., Rector, A.L., Stevens, R., Moulton, G., Horridge, M., Wang, H., Seidenberg, J.: Putting OWL in order: Patterns for sequences in OWL. In: Proceedings of OWLED (2006)
17. Euzenat, J., Shvaiko, P.: Ontology matching. Springer, Heidelberg (2007)
18. Howar, F., Merten, M., Neubauer, J., Steffen, B.: Introduction to automata learning. In: Bernardo, M., Issarny, V. (eds.) SFM 2011. LNCS, vol. 6659, pp. 350–392. Springer, Heidelberg (2011)
19. Flores-Cortés, C.A., Blair, G.S., Grace, P.: An adaptive middleware to overcome service discovery heterogeneity in mobile ad hoc environments. IEEE Distributed Systems Online 8(7) (2007)
20. Foster, H., Uchitel, S., Magee, J., Kramer, J.: LTSA-WS: a tool for model-based verification of web service compositions and choreography. In: Proceedings of ICSE (2006)
21. Grace, P., Blair, G.S., Samuel, S.: ReMMoC: A reflective middleware to support mobile client interoperability. In: Chung, S., Schmidt, D.C. (eds.) CoopIS 2003, DOA 2003, and ODBASE 2003. LNCS, vol. 2888, pp. 1170–1187. Springer, Heidelberg (2003)
22. Grace, P., Georgantas, N., Bennaceur, A., Blair, G., Chauvel, F., Issarny, V., Paolucci, M., Saadi, R., Souville, B., Sykes, D.: The connect architecture. In: Bernardo, M., Issarny, V. (eds.) SFM 2011. LNCS, vol. 6659, pp. 350–392. Springer, Heidelberg (2011)
23. Green Jr., P.: Protocol conversion. IEEE Transactions on Communications 34(3) (March 1986)
24. Hoare, C.A.R.: Communicating sequential processes. Communications of the ACM (CACM) 21(8) (1978)
25. Howar, F., Jonsson, B., Merten, M., Steffen, B., Cassel, S.: On handling data in automata learning - considerations from the connect perspective. In: Margaria, T., Steffen, B. (eds.) ISoLA 2010. LNCS, vol. 6416, pp. 221–235. Springer, Heidelberg (2010)

26. Inverardi, P., Spalazzese, R., Tivoli, M.: Application-layer connector synthesis. In: Bernardo, M., Issarny, V. (eds.) SFM 2011. LNCS, vol. 6659, pp. 350–392. Springer, Heidelberg (2011)
27. Issarny, V., Caporuscio, M., Georgantas, N.: A Perspective on the Future of Middleware-based Software Engineering. In: Proceedings of FOSE 2007 (2007)
28. Issarny, V., Steffen, B., Jonsson, B., Blair, G., Grace, P., Kwiatkowska, M., Calinescu, R., Inverardi, P., Tivoli, M., Bertolino, A., Sabetta, A.: CONNECT Challenges: Towards Emergent Connectors for Eternal Networked Systems. In: Proceedings of the 14th ICECCS (2009)
29. Kumar, R., Nelvagal, S., Marcus, S.I.: A discrete event systems approach for protocol conversion. Discrete Event Dynamic Systems 7 (June 1997)
30. Lam, S.S.: Protocol conversion. IEEE Transaction Software Engineering 14(9) (1988)
31. Limam, N., Ziembicki, J., Ahmed, R., Iraqi, Y., Li, T., Boutaba, R., Cuervo, F.: Osda: Open service discovery architecture for efficient cross-domain service provisioning. Computer Communications 30(3) (2007)
32. Liskov, B.H., Wing, J.M.: A behavioral notion of subtyping. ACM Trans. on Prog. Lang. and Syst. (1994)
33. Magee, J., Kramer, J.: Concurrency: State models and Java programs. Wiley, Hoboken (2006)
34. Mehta, N.R., Medvidovic, N., Phadke, S.: Towards a taxonomy of software connectors. In: Proceedings of ICSE (2000)
35. Menge, F.: Enterprise Service Bus. In: Free and Open Source Software Conference (2007)
36. Nakazawa, J., Tokuda, H., Edwards, W.K., Ramachandran, U.: A bridging framework for universal interoperability in pervasive systems. In: Proceedings of ICDCS (2006)
37. Nezhad, H.R.M., Benatallah, B., Martens, A., Curbera, F., Casati, F.: Semi-automated adaptation of service interactions. In: Proceedings of WWW (2007)
38. Nitto, E.D., Rosenblum, D.S.: Exploiting adls to specify architectural styles induced by middleware infrastructures. In: Proceedings of ICSE (1999)
39. Okumura, K.: A formal protocol conversion method. In: Proceedings of SIGCOMM (1986)
40. (OMG): COM/CORBA interworking specification Part A & B (1997)
41. Paolucci, M., Kawamura, T., Payne, T.R., Sycara, K.P.: Semantic matching of web services capabilities. In: Horrocks, I., Hendler, J. (eds.) ISWC 2002. LNCS, vol. 2342, p. 333. Springer, Heidelberg (2002)
42. Ponnekanti, S., Fox, A.: Interoperability among independently evolving web services. In: Jacobsen, H.-A. (ed.) Middleware 2004. LNCS, vol. 3231, pp. 331–351. Springer, Heidelberg (2004)
43. Raverdy, P.G., Issarny, V., Chibout, R., de La Chapelle, A.: A multi-protocol approach to service discovery and access in pervasive environments. In: Proceedings of MobiQuitous. IEEE Computer Society, Los Alamitos (2006)
44. Román, M., Campbell, R.H., Kon, F.: Reective middleware: From your desk to your hand. IEEE Distributed Systems Online 2(5) (2001)
45. Spalazzese, R., Inverardi, P.: Mediating Connector Patterns for Components Interoperability. In: Babar, M.A., Gorton, I. (eds.) ECSA 2010. LNCS, vol. 6285, pp. 335–343. Springer, Heidelberg (2010)
46. Spalazzese, R., Inverardi, P., Issarny, V.: Towards a formalization of mediating connectors for on the y interoperability. In: Proceedings of WICSA/ECSA (2009)

47. Spitznagel, B., Garlan, D.: A compositional formalization of connector wrappers. In: Proceedings of ICSE (2003)
48. Stollberg, M., Cimpian, E., Mocan, A., Fensel, D.: A semantic web mediation architecture. In: Proceedings of CSWWS (2006)
49. Studer, R., Benjamins, V.R., Fensel, D.: Knowledge engineering. Data & Knowledge Engineering (1998)
50. Taylor, R.N., Medvidovic, N., Dashofy, E.M.: Software architecture: foundations, theory, and practice. Wiley, Hoboken (2009)
51. Vaculín, R., Sycara, K.P.: Towards automatic mediation of OWL-S process models. In: Proceedings of ICWS (2007)
52. Yellin, D.M., Strom, R.E.: Protocol specifications and component adaptors. ACM Trans. Program. Lang. Syst. 19(2) (1997)
53. Zhu, F., Mutka, M., Ni, L.: Service discovery in pervasive computing environments. Pervasive Computing (2005)

Introduction to Active Automata Learning from a Practical Perspective*

Bernhard Steffen, Falk Howar, and Maik Merten

TU Dortmund University, Chair for Programming Systems,
Dortmund, D-44227, Germany
{steffen,falk.howar,maik.merten}@cs.tu-dortmund.de

Abstract. In this chapter we give an introduction to active learning of Mealy machines, an automata model particularly suited for modeling the behavior of realistic reactive systems. Active learning is characterized by its alternation of an exploration phase and a testing phase. During exploration phases so-called membership queries are used to construct hypothesis models of a system under learning. In testing phases so-called equivalence queries are used to compare respective hypothesis models to the actual system. These two phases are iterated until a valid model of the target system is produced.

We will step-wisely elaborate on this simple algorithmic pattern, its underlying correctness arguments, its limitations, and, in particular, ways to overcome apparent hurdles for practical application. This should provide students and outsiders of the field with an intuitive account of the high potential of this challenging research area in particular concerning the control and validation of evolving reactive systems.

1 Motivation

Interoperability remains a fundamental challenge when connecting heterogeneous systems [10]. The Connect Integrated Project [35] aims at overcoming the interoperability barrier by synthesizing required Connectors on the fly in five steps [5,21]: (i) extracting knowledge from, (ii) learning about and (iii) reasoning about the interaction behavior of networked systems, as a basis for (iv) synthesizing Connectors [33,34], and subsequently (v) generating and deploying them for immediate use [9].

This chapter focuses on the foundations for step (ii), namely on techniques for leveraging and enriching the extracted knowledge by means of experimentation with the targeted components. Central are here advanced techniques for *active* automata learning [3,38,4,31,50], which are designed for optimally aggregating, and where necessary completing, the observed behavior.

Characteristic for active learning automata learning is its iterative alternation between a "testing" phase for completing the transitions relation of the model aggregated from the observed behavior, and an equivalence checking phase,

* This work is supported by the European FP 7 project CONNECT (IST 231167).

M. Bernardo and V. Issarny (Eds.): SFM 2011, LNCS 6659, pp. 256–296, 2011.

which either signals success or provides a counterexample, i.e., a behavior that distinguishes the current aggregate (called hypothesis) from the system to be learned. In practice, this second phase must typically be (approximately) realized via testing. This is the reason for the learning approach neither to be correct nor complete in practice. However, it can be proven that it optimally aggregates the behavior optimal in the following sense: hypothesis models are guaranteed to be the most concise (state-minimal) consistent representation of the observed behavior.

This technique, which originally has been introduced for dealing with formal languages, works very well also for reactive systems, whenever the chosen interpretation of the stimuli and reactions leads to a deterministic language. For such systems, active automata learning can be regarded as *regular extrapolation*, i.e., as a technique to construct the "best" regular model being consistent with the observations made. This is similar to the well-known polynomial extrapolation, where polynomials are used instead of finite automata, and functions instead of reactive systems. And like there, the quality, not the applicability, of extrapolation depends on the structure of the considered system behavior. However, due to the enormous degree of freedom inherent in reactive systems, automata learning is computationally much more expensive than polynomial extrapolation. Thus the success of automata learning in practice very much depends on the optimizations employed to exploit the specific profile of the system to be learned [31,50]. One important step in the direction was the generalization of the modeling structure from deterministic automata to Mealy machines [31,45,32,52]. We will therefore consider this setup throughout this chapter.

Outline: In this chapter we will review the foundations of active learning for Mealy machines, which have proven to be an adequate modeling formalism for reactive systems in practice.

We will start in Section 2 by formally introducing Mealy machines and transferring the idea of the Nerode-relation from languages to the world of Mealy machines. This will provide us with a mechanism for distinguishing states of an unknown system under learning. Finally, we will revisit the idea of partition refinement, using a simple minimization algorithm for Mealy machines as an example.

In Section 3 we will then exactly define the scenario of active learning and discuss how the ideas from Section 2 can be put together conceptually in order to infer models from black-box systems. This scheme will be used in Section 4 when we present an intuitive algorithm that uses a direct on-the-fly construction approach to learning. In Section 5 we present a modified L^* learning algorithm for Mealy machines that uses the data-structures and building blocks usually used in active learning literature.

Finally, we will discuss briefly the challenges to be faced when using active learning in real-world scenarios in Section 6 and present a framework for automata learning in Section 7, before we conclude in Section 8. A more detailed account of practical challenges is given in [30,36]. Section 9 contains pointers to literature for the interested reader and in Section 10 you will find some exercises based on the concepts presented in this chapter.

2 Modeling Reactive Systems

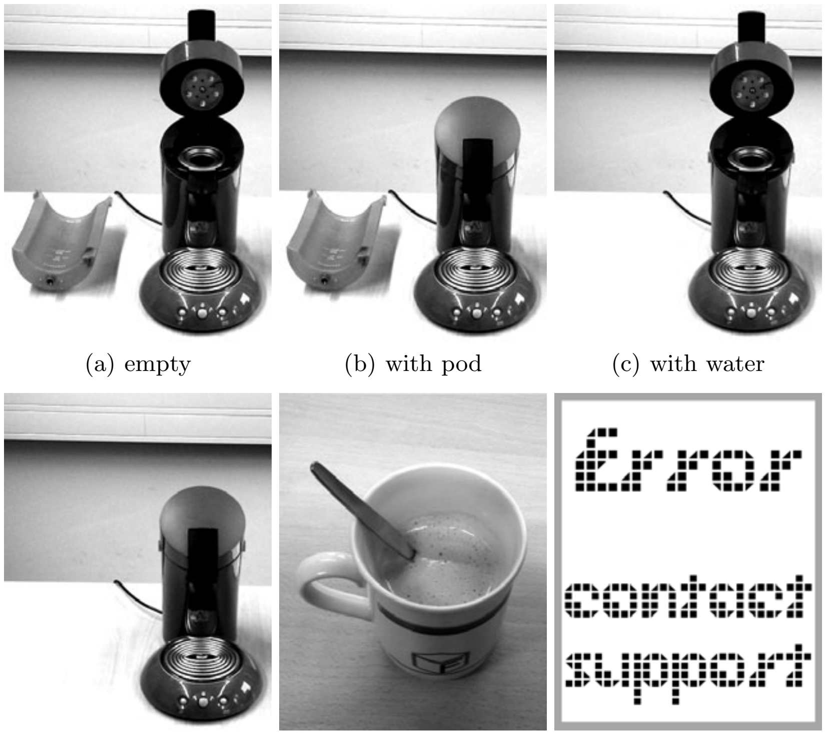
In this section we will discuss how to model reactive systems formally and introduce Mealy machines as an adequate formalism for modeling systems with input and output. We will see that Mealy machines can be given semantics in terms of *runs* in the same way as finite automata are interpreted as representations of formal languages. Subsequently, we exploit this similarity to a Myhill/Nerode-like theorem for the Mealy scenario. This will allow us to define canonical models and provides us with a handle to construct Mealy machines from sets of runs. Finally, we will present a minimization algorithm for Mealy machines and thereby revisit the concept of partition refinement, the characteristic algorithmic pattern underlying active learning.

Most of the systems we are using every day – imagine a telephony system – can be seen as reactive systems: These systems usually (almost) never terminate and interact with their environment, e.g., with a user or another system. They expose a set of input actions to their environment and on a specific input these systems will produce some output: In a telephony system, e.g., after dialing a number, you may hear a ringing tone. Alternatively, you might also hear a busy tone. From a user's perspective this behavior of the system is not (input) deterministic, (i.e., the reaction to the same input (sequence) leads to two different reactions (outputs)) although, from a more detailed perspective it is not: including additional information on the "state" of the system will expose the causal prerequisites of hearing a ringing tone or busy tone. In this particular case, we even know the causal connections: when we attempt to call someone who is in a call already, we will hear the busy tone. Thus the apparent (input) non-determinism can be overcome by considering the larger context including the activities that led to the called party being already on a call.

Active automata learning very much depends on the system under learning to be (input) deterministic. Usually this is not too much of a restriction as indicated above: apparently non-deterministic practical (man made) systems can usually be "made" deterministic by adding detail (refinement). Otherwise, the system would not be controllable, which often is considered a reliability problem.

Example 1 (A coffee machine). Let us consider a very simple reactive system: a coffee machine. This machine has an assessable user interface, namely a button which starts the production of delicious coffee. However, before the production of this precious fluid can commence, a water tank (filled with water) and a coffee pod have to be put in place. After every cup of coffee produced, the machine has to be cleaned, which involves the removal of all expendables. Thus the operations possible on the machine are "*water*" (fill the water tank), "*pod*" (provide a fresh coffee pod), "*clean*" (remove all expendables) and "*button*" (start the production of coffee).

One single flaw that escaped product testing, however, is that the machine will immediately enter an error state on any mishandling. If, e.g., the button for coffee production is pressed before a complete set of expendables is filled in, an error will be signaled that cannot be overcome using the conventional interaction operations described above. This explains the lukewarm reception by consumers and in turn the affordable price of the machine.



(a) empty (b) with pod (c) with water

(d) with pod and water (e) success (f) error

Fig. 1. The illustrated state space of the coffee machine

The state space of the machine is readily observable (see Fig. 1), as is the output produced: the machine can be "OK" ("✓") with a user interaction, produce coffee ("☕"), or express its dissatisfaction with the way it is operated by signaling an error ("✷"). □

2.1 Mealy Machines

Mealy machines are a variant of automata which distinguish between an input alphabet and an output alphabet. Characteristic for Mealy machines is that inputs are always enabled (in other words the transition function is totally defined for all input symbols), and that their response to an input (sequence) is uniquely determined (this property is called input determinism). Both properties fit the requirements of a large class of (reactive) systems very well.

We will use Mealy machines throughout this chapter. It should, however, be noted that there is a very close relationship between Mealy machines and deterministic finite automata: Mealy machines can be regarded as deterministic finite automata over the union of the input alphabet and an output alphabet with just one rejection state, which is a sink, or more elegantly, with a partially defined transition relation. In fact, considering partially defined transition relations provides a close analogy, as Mealy machines do not distinguish between accepting and rejecting states. They distinguish runs according to their output. Semantically this means that these automata define prefix closed languages, an adequate choice when modeling the reactive behavior of a system, because one cannot observe a long run without first seeing its prefixes.

More formally, we assume a set of input actions Σ and a set of outputs Ω, and we refer as usual to sequences of inputs (or outputs) $w = \alpha_1 \ldots \alpha_n$, where $\alpha_i \in \Sigma$, as *words*, which can be concatenated, as well as split into prefixes and suffixes in the same way as known from language theory: we write $w = uv$ to denote that w can be split into a prefix u and a suffix v, or – reversely – that u and v can be concatenated to w. Sometimes, when we want to emphasize the concatenation, we write $u \cdot v$, and we denote the empty word by ϵ.

Let us now define a Mealy machine:

Definition 1. *A Mealy machine is defined as a tuple $\mathcal{M} = \langle S, s_0, \Sigma, \Omega, \delta, \lambda \rangle$ where*

- *S is a finite nonempty set of* states *(be $n = |S|$ the size of the Mealy machine),*
- *$s_0 \in S$ is the* initial state*,*
- *Σ is a finite* input alphabet*,*
- *Ω is a finite* output alphabet*,*
- *$\delta : S \times \Sigma \to S$ is the* transition function*, and*
- *$\lambda : S \times \Sigma \to \Omega$ is the* output function*.*

Intuitively, a Mealy machine evolves through states $s \in S$, and whenever one applies an input symbol (or action) $\alpha \in \Sigma$, the machine moves to a new state according to $\delta(s, \alpha)$ and produces an output according to $\lambda(s, \alpha)$. □

We write $s \xrightarrow{\alpha/o} s'$ to denote that on input symbol α the Mealy machine moves from state s to state s' producing output symbol o. We will denote the straightforward inductive extensions of $\delta : S \times \Sigma \to S$ and $\lambda : S \times \Sigma \to \Omega$ to deal with words in the second component with δ^* and λ^*, respectively. $\delta^* : S \times \Sigma^* \to S$ and $\lambda^* : S \times \Sigma^* \to \Omega$ are formally defined by $\delta^*(s, \epsilon) = s$ and $\delta^*(s, \alpha w) = \delta^*(\delta(s, \alpha), w)$; by $\lambda^*(s, \epsilon) = \varnothing$ and $\lambda^*(s, w\alpha) = \lambda(\delta^*(s, w), \alpha)$ respectively.

Example 2 (Modeling the coffee machine). We can specify the behavior of the coffee machine from Example 1 as the Mealy machine $\mathcal{M}_{cm} = \langle S, s_0, \Sigma, \Omega, \delta, \lambda \rangle$, where

- $S = \{a, b, c, d, d', e, f\}$
- $s_0 = a$
- $\Sigma = \{water, pod, button, clean\}$
- $\Omega = \{$✓, ☕, ✱$\}$

The transition- and output-function are defined according to the model shown in Fig. 2. In this example specification, we use two states d and d' differing wrt. the order in which water and pod have been filled into the machine. □

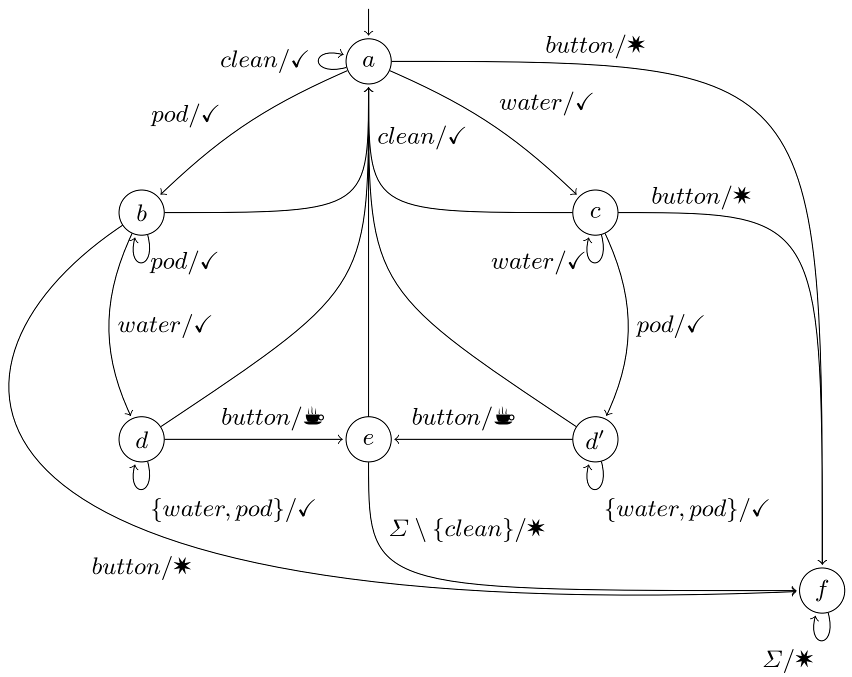


Fig. 2. Mealy specification of the coffee machine

The concrete behavior of a Mealy machine when processing a sequence of inputs $\alpha_1\alpha_2\ldots\alpha_n$ has the pattern of an alternating sequence of input and output symbols $\alpha_1 o_1 \alpha_2 o_2 \ldots \alpha_n o_n$. It turns out, however, that Mealy machines can be fully characterized in terms of their *runs*, which abstract from all the intermediate outputs and simply record the the final output. This means that the following semantic functional $[\![M]\!] : \Sigma^* \rightarrow \Omega$ defined by $[\![M]\!](w) = \lambda^*(s_0, w)$ faithfully captures the behavioral semantics of Mealy machines (see Theorem 1). In particular, we will see that the corresponding notion of semantic equivalence $\mathcal{M} \equiv \mathcal{M}'$ defined by $[\![\mathcal{M}]\!] = [\![\mathcal{M}']\!]$ is the leading notion in the following development.

Example 3 (Runs of Mealy machines). The run

$$\langle \textit{water pod button clean button},\ ✷\rangle$$

is in $[\![\mathcal{M}_{cm}]\!]$, while the run

$$\langle \textit{water button clean},\ \checkmark\rangle$$

is not, because in $\mathcal{M}_{cm}$, once a run passes state f no other output than ✷ will be produced. □

2.2 Regularity

In this section we will characterize which functionals $P : \Sigma^* \to \Omega$ are the semantics of some Mealy machine. Key to this characterization is the following notion of equivalence induced by P on input words which resembles the well-known Nerode relation for formal languages [44]:

Definition 2 (Equivalence of words wrt. P). *Two words $u, u' \in \Sigma^*$ are equivalent wrt. $\equiv_P$, iff for all continuations $v \in \Sigma^*$, the concatenated words uv and $u'v$ are mapped to the same output by P:*

$$u \equiv_P u' \;\Leftrightarrow\; (\forall v \in \Sigma^*.\; P(uv) \;=\; P(u'v)).$$

We write $[u]$ to denote the equivalence class of u wrt. $\equiv_P$. □

Obviously, $\equiv_P$ is an equivalence relation: equality is reflexive, symmetric, and transitive. Also, we observe that every Mealy machine $\mathcal{M}$ for P refines such a relation $\equiv_P$: Two words $u, u' \in \Sigma^*$ leading to the same state have to be in the same class of $\equiv_P$ as the future behavior of $\mathcal{M}$ for both words is identical.

Example 4 (Equivalence of words wrt. P). In our example model of $\mathcal{M}_{cm}$ from Example 2 and Figure 2, the following three words (among others) are equivalent wrt. $\equiv_{[\![\mathcal{M}_{cm}]\!]}$:

$$\begin{aligned} & \textit{water pod} && (1)\\ \equiv_{[\![\mathcal{M}_{cm}]\!]}\; & \textit{water water pod} && (2)\\ \equiv_{[\![\mathcal{M}_{cm}]\!]}\; & \textit{pod pod water} && (3)\end{aligned}$$

For (1) and (2) it is obvious, because (1) and (2) lead to the same state (d). The word (3), on the other hand, leads to a different state (d'). However, we defined the equivalence relation on Σ^*, and not on $\mathcal{M}_{cm}$. We leave it to the reader to retrace that there does not exists a possible continuation of (1) and (3) in Σ^*, which proves both words inequivalent. □

This is already sufficient to prove our Characterization Theorem as a straightforward adaption of the Myhill/Nerode theorem for regular languages and deterministic finite automata (DFA) [26,44].

Theorem 1 (Characterization Theorem). *A mapping $P : \Sigma^* \to \Omega$ is a semantic functional for some Mealy machine iff $\equiv_P$ has only finitely many equivalence classes (finite index).*

Proof. ($\Rightarrow$): Let $\mathcal{M}$ be an arbitrary Mealy machine. Then we must show that $\equiv_{[\![M]\!]}$ has finite index. This follows directly from that fact that all input words that lead to the same state of the $\mathcal{M}$ are obviously equivalent, which limits the index by the number of state of $\mathcal{M}$.

($\Leftarrow$): Consider the following definition of Mealy machine $\mathcal{M}_P = \langle S, s_0, \Sigma, \Omega, \delta, \lambda \rangle$:

- S is given by the classes of $\equiv_P$.
- s_0 is given by $[\epsilon]$.
- the transition function is defined by $\delta([w], \alpha) = [w\alpha]$.
- the output function can be defined as $\lambda([w], \alpha) = o$, where $P(w\alpha) = o$.

Then it is straightforward to verify that $\mathcal{M}_P$ is a well-defined Mealy machine with semantic functional P, i.e., with $[\![\mathcal{M}_P]\!] = P$. □

In analogy to classical language theory, we will call a mapping $P : \Sigma^* \rightarrow \Omega$ regular whenever there exists a corresponding Mealy machine $\mathcal{M}_P$, or, equivalently, whenever $\equiv_P$ has finite index. In this situation it is easy to establish a corresponding variant of the famous Pumping Lemma (cf. [26]), again in full analogy to classical language theory:

Proposition 1 (Bounded reachability). *Every state of a minimal Mealy machine with n states has an access sequence, i.e., a path from the initial state to this state, of length at most $n - 1$. Every transition of this Mealy machine can be covered by a sequence of length at most n from the initial state.*

A more careful look at the Mealy machine $\mathcal{M}_P$ constructed above reveals that it is indeed the up to isomorphism unique state-minimal Mealy machine with semantic functional P, as two input words can obviously only lead to the same state if they are equivalent wrt. $\equiv_P$. This makes $\mathcal{M}_P$ the canonical model for representing P.

Example 4 shows that a Mealy machine can have more than one state per class of $\equiv_{[\![\mathcal{M}]\!]}$. In the following, we will investigate when and how we can effectively transform such Mealy machines into canonical form. Please note that the construction in the proof of the Characterization Theorem is not effective as it is in general not possible to compute $\equiv_P$ just from P. We will see that there is a smooth transition from variants of minimization algorithms to the underlying pattern of L^*, Angluin's seminal active learning algorithm.

2.3 Canonical Mealy Machines

For an arbitrary Mealy machine, we will usually have no information about the index of $\equiv_{[\![\mathcal{M}]\!]}$, the classes of $\equiv_{[\![\mathcal{M}]\!]}$, or how single states correspond to classes of $\equiv_{[\![\mathcal{M}]\!]}$. From Theorem 1 we know, however, that all words leading to the same state, have to be in the same class of $\equiv_{[\![\mathcal{M}]\!]}$, and that there exists a Mealy machine whose states directly correspond to the equivalence classes of $\equiv_{[\![\mathcal{M}]\!]}$. Unfortunately, trying the minimize a given Mealy machine by merging some states whose access sequences are $\equiv_{[\![\mathcal{M}]\!]}$-equivalent has two drawbacks:

- it may destroy the well-definedness of the transitions function δ (which could be overcome by generalizing the notion for Mealy automaton to allow for transitions relations), and, much worse,
- proving the equivalence of access sequences is in general quite hard. In the setting of active learning of black box system (cf. Section 6) it is even undecidable in general.

However there is an alternative way, which in addition to its elegance and efficiency, paves the way to active automata learning: *partition refinement.* Rather than collapsing a too fine partition on access sequences (given here by the states of a Mealy machine), partition refinement works by iteratively refining too coarse partitions (initially typically the partition with just one class) based on so called *distinguishing suffixes*, i.e., suffixes that witness the difference of two access sequences.

Remark. Both approaches, the collapsing-based approach and the refinement-based approach, iteratively compute fixed points on the basis of $[\![\mathcal{M}]\!]$: collapsing the smallest, and refining the greatest. As the fixed point is unique in this case, both approaches would lead to the same result.

Theorem 1 and its underlying construction of $\mathcal{M}_P$ provide the conceptual backbone for all the following variants of partition refinement algorithms. The following notion is important:

Definition 3 (k-distinguishability). *Two states $s, s' \in S$ of some Mealy machine $\mathcal{M}$ are k-distinguishable, iff there is a word $w \in \Sigma^*$ of length k or shorter, for which $\lambda^*(s, w) \neq \lambda^*(s', w)$.* □

Intuitively, two states are k-distinguishable, if starting from both states we can produce different outputs when processing the same suffix within k steps. To ease readability, we introduce *exact* k-distinguishability, denoted by $k^=$, for states that are k-distinguishable, but not $(k-1)$-distinguishable.

As a general prerequisite for the following developments, we will assume that we can effectively ask so-called *membership queries* (a central notion in active learning), i.e., that there is a so-called *membership oracle* which returns $[\![\mathcal{M}]\!](w)$ in constant time, whenever it is asked for w, and we will measure the efficiency of the following approaches to constructing canonical Mealy machines just in the number of required membership queries. We denote the canonical representation of some Mealy machine $\mathcal{M}$ by $\mathcal{C}([\![\mathcal{M}]\!])$.

Constructing $[\![\mathcal{M}]\!]$ for words of sufficient length: Knowing an upper approximation N of the index of $\equiv_{[\![\mathcal{M}]\!]}$ is already sufficient to effectively construct $\mathcal{C}([\![\mathcal{M}]\!])$ in exponential time. Due to Proposition 1

– all states of $\mathcal{C}([\![\mathcal{M}]\!])$ are N-distinguishable, and
– the set Σ^N of all words of length up to N is guaranteed to contain an access sequence to every state and to cover every transition of $\mathcal{C}([\![\mathcal{M}]\!])$.

With this knowledge, $\mathcal{C}([\![\mathcal{M}]\!])$ can be constructed along the lines presented in the proof of Theorem 1, leading to a combinatorial $O(|\Sigma|^{2N})$ algorithm.

Using access sequences from $\mathcal{M}$: If we now, additionally, take advantage of the knowledge of the n states of the Mealy machine to be minimized, the complexity of the algorithm sketched above immediately reduces to $O(n|\Sigma| \cdot |\Sigma|^N)$, as we are able to access every state and to cover every transition of $\mathcal{C}([\![\mathcal{M}]\!])$ just by looking at the n states and the $n|\Sigma|$ transitions of that Mealy machine.

Using access sequences and suffixes from $\mathcal{M}$: This naive algorithm can be drastically improved based on the following elementary observation: Whenever two states s_1 and s_2 are $(k+1)$-distinguishable then they each have a α-successor ${s'}_1$ respectively ${s'}_2$ (for some $\alpha \in \Sigma$) such that ${s'}_1$ and ${s'}_2$ are k-distinguishable. This suggests the following inductive characterization of k-distinguishability:

- no states are 0-distinguishable, and
- two states s_1 and s_2 are $(k+1)$-distinguishable iff there exists an input symbol $\alpha \in \Sigma$ such that $\lambda(s_1, \alpha) \neq \lambda(s_2, \alpha)$ or $\delta(s_1, \alpha)$ and $\delta(s_2, \alpha)$ are k-distinguishable.

which directly leads to an algorithm that iteratively computes k-distinguishability for increasing k until stability, i.e., until the set of exactly k-distinguishable states is empty, in analogy to the original algorithm by Hopcroft for minimizing deterministic finite automata [25]. It is straightforward to deduce that each level of k-distinguishability can be done in $O(n|\Sigma|)$, i.e., by processing every transition once, and that k will never exceed n. Thus we arrived at an $O(n^2|\Sigma|)$ algorithm, for which corresponding pseudocode is shown in Algorithm 1.

Example 5 (Partition refinement). Assume the Mealy machine from Figure 2 as an input to Algorithm 1. We start by computing the initial partition P_1:

$$P_1 = \{a, b, c\}, \{d, d'\}, \{e\}, \{f\}$$

where "*clean*" distinguishes e from f, and "*button*" distinguishes, e.g., a from d. In the second step, we will generate:

$$P_2 = \{a\}, \{b\}, \{c\}, \{d, d'\}, \{e\}, \{f\}$$

where "*water*" and "*pod*" distinguish a, b and c: The "*water*"-successor of a and c is c, while for b it is d. The "*pod*"-successor of a and b is b, while for c it is d'.

Then, however, in the next step, we will not be able to further refine P_2. We can merge d and d' and get the Mealy machine depicted in Figure 3. □

The correctness of this algorithm follows a very well-known three-step proof pattern:

- **Invariance:** The number of equivalence classes obtained during the partition refinement process never exceeds the index of $\equiv_{[\![\mathcal{M}]\!]}$. This follows from the fact that only distinguishable states are split.
- **Progress:** Before the final partition is reached, it is guaranteed that the investigation of all transitions of $\mathcal{M}$ will suffice to split at least one equivalence class. This follows from the inductive characterization of distinguishability in terms of k-distinguishability.
- **Termination:** The partition refinement process terminates after at most index of $\equiv_{[\![\mathcal{M}]\!]}$ many steps. This is a direct consequence of the just described properties invariance and progress.

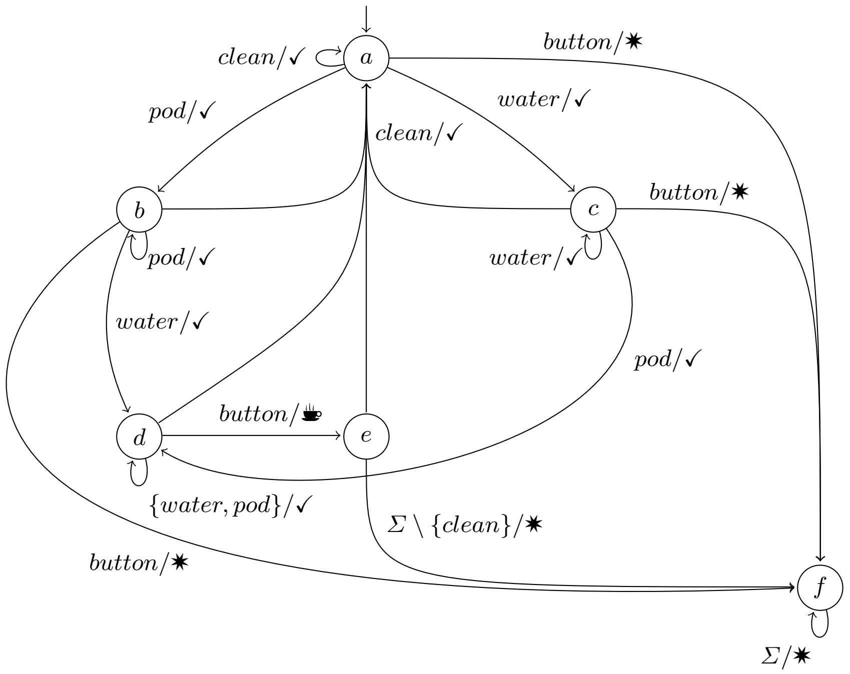


Fig. 3. Minimal Mealy machine of the coffee machine

Algorithm 1. Compute partition on set of states

Input: A Mealy machine $\mathcal{M} = \langle S, s_0, \Sigma, \Omega, \delta, \lambda \rangle$

Output: A partition P on S, the set of states of the Mealy machine

```
 1: $i := 1$
 2: put all $s \in S$ with the same $\lambda$ valuation into the same class $p$ of partition $P_i$
 3: loop
 4:    for all $p \in P_i$ do
 5:       for all $s \in p$ do
 6:          construct mapping $sig: \Sigma \rightarrow P_i$:
 7:                $sig(\alpha) = p'$ such that $\delta(s, \alpha) \in p'$
 8:          $S_{sig} := S_{sig} \cup s$
 9:       end for
10:       $P_{i+1} := \bigcup_{sig} S_{sig}$
11:    end for
12:    if $P_i = P_{i+1}$ then
13:       return $P_i$
14:    end if
15:    $i := i + 1$
16: end loop
```

In oder to better understand the essential difference between minimization and active learning, let $\mathcal{M} = \langle S, s_0, \Sigma, \Omega, \delta, \lambda \rangle$ and $\mathcal{M}' = \langle S', s_0', \Sigma, \Omega, \delta', \lambda' \rangle$ be two Mealy machines with shared alphabets. Then we call a surjective function $f_k : S \to S'$ *existential k-epimorphism* between $\mathcal{M}$ and $\mathcal{M}'$, if for all $s' \in S'$, $s \in S$ with $f_k(s) = s'$, and $\alpha \in \Sigma$ we have: $f_k(\delta(s, \alpha)) = \delta'(s', \alpha)$ and all state that are mapped by f_k to the same state of $\mathcal{M}'$ are not k-distinguishable.

It is straightforward to establish that all intermediate models arising during the partition refinement process are images of the considered Mealy machine under a k-epimorphism, where k is the number of times all transitions have been investigated. In the next section, we will establish a similar notion of epimorphism which fits the active learning scenario. Its difference to k-epimorphism will help us to maintain as much of the argument discussed here as possible and to formally pinpoint some essentially differences between minimization and learning.

More generally, the pattern of this algorithm and its correctness proof will guide us in the following sections, where we are going to develop an algorithm to infer a canonical Mealy machine from black box systems by experimentation/testing. In contrast to this section, where we exploited knowledge in terms of a given realizing Mealy machine, or at least of the number of states of such a machine, we will start by assuming an ideal, indeed quite unrealistic, but in the community accepted operation: the so-called *equivalence oracles*. They can be queried in terms of so-called *equivalence queries* asking for the semantic equivalence of the already computed hypothesis model and the black box systems, and in case of failure provide evidence in terms of a counterexample. We will see that under these assumption it is possible to also learn the canonical Mealy machine for regular black box systems with polynomial complexity measured in membership and equivalence queries.

3 Construction of Models from Black-Box Systems

In principle, we are concerned with the same problem as in the previous section: the construction of a canonical model for some Mealy machine $\mathcal{M}$. Only the frame conditions changed. Rather than having $\mathcal{M} = \langle S, s_0, \Sigma, \Omega, \delta, \lambda \rangle$ at hand, we only have limited access to the resources: active learning algorithms classically use two kinds of queries to gather information about a black-box system under learning (SUL) – the notion is meant to remind of the term System Under Test (SUT) used by the testing community – which are assumed to be realized via two corresponding oracles, resembling a “teacher” who is capable of answering these queries appropriately and correctly according to the minimally adequate teacher (MAT) model [3]. These queries, which have been sketched already in the previous section, are the so-called:

Membership Queries retrieving behavioral information about the target system. Consisting of sequences of system inputs, they actively trigger behavioral outputs which are collected and analyzed by the learning algorithm. Membership queries are used to construct a hypothesis model, which is then subject to validation by means of the second kind of queries, the equivalence queries.

We write $\text{mq}(w) = o$ to denote that executing the query $w \in \Sigma^*$ on SUL leads to the output o, meaning that $\lambda^*_{SUL}(q_0, w) = o$. In practice, membership queries correspond to single test runs executed on a system to be learned.

Equivalence Queries determining whether the learned hypothesis is a faithful representation of the target system. If the equivalence oracle handling the equivalence query finds diverging behavior between the learned hypothesis and the SUL, a counterexample will be produced, which is used to refine the hypothesis model with a next iteration of the learning algorithm.

We write $\text{eq}(\mathcal{H}) = \bar{c}$ to denote that the equivalence query for $\mathcal{H}$ returned a counterexample $\bar{c} \in \Sigma^*$ with $\lambda^*_{\mathcal{H}}(s_0, \bar{c}) \neq \text{mq}(\bar{c})$. In practice, equivalence queries can typically not be realized. We will discuss this problem and possible solutions in Section 6. Equivalence queries are, however, an elegant concept for structuring active learning algorithms.

In the following, we will study this classical active learning scenario, before we will discuss its limitations, associated problems and ways to reach practicality in Section 6. In particular we will see that based on these two kinds of queries active learning algorithms such as L^*_M effectively create canonical automata models of the SUL, whenever the SUL is regular (cf. Section 2.2).

The high-level algorithmic patterns underlying most active learning algorithms is shown in Fig. 4: active learning proceeds in rounds, generating a sequence of so-called hypothesis models by alternating test-based exploration on the basis of membership queries and equivalence checking using the equivalence oracle. Counterexamples resulting from failing equivalence checks are used to steer the next round of local exploration. The first step shown in Fig. 4, the setup of a learning algorithm from some input requirements, will briefly be discussed in Section 6.

Following the partition-refinement pattern we used in Section 2.3 to minimize Mealy machines, inference starts with the one state hypothesis automaton that treats all words over the considered alphabet (of elementary observations) alike and refines this automaton on the basis of the query results, iterating test-based exploration steps and the equivalence checking steps. Here, the dual way of how states are characterized (and distinguished) is central:

– by words reaching them. A prefix-closed set $\mathcal{S}p$ of words, reaching each state exactly once, defines a spanning tree of the automaton. This characterization aims at providing exactly one representative element from each class of $\equiv_P$ on the SUL. Active learning algorithms incrementally construct such a set $\mathcal{S}p$.

  Prefix-closedness will guarantee that the constructed set is a "spanning tree" of the unknown Mealy machine. Extending this spanning tree to contain also all one-letter continuations of words in $\mathcal{S}p$ will result in a tree covering all the transitions of the Mealy machine. We will denote the set of all one-letter continuations that are not already contained in $\mathcal{S}p$ by $\mathcal{L}p$.
– by their future behavior wrt. a dynamically increasing vector of strings from Σ^*. This vector $\langle d_1 \dots d_k \rangle$ will be denoted by $\mathcal{D}$, for "distinguishing suffixes".

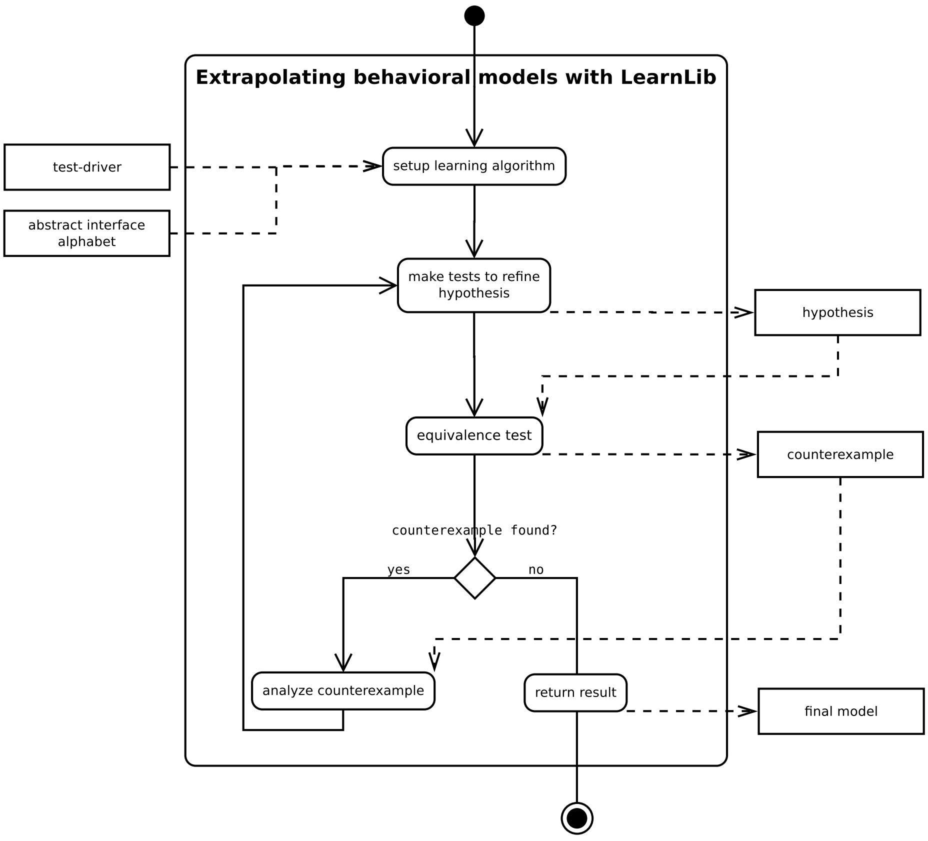


Fig. 4. Structure of Extrapolation Algorithms (modeled in XPDD [37]). Square boxes on the left hand side denote inputs, on the right hand side they denote outputs.

> The corresponding future behavior of a state, here given in terms of its access sequence $u \in \mathcal{S}p$, is the output vector $\langle \mathrm{mq}(u \cdot d_1) \dots \mathrm{mq}(u \cdot d_k) \rangle \in \Omega^k$, which leads to an upper approximation of the classes of $\equiv_{[\![SUL]\!]}$. Active learning incrementally refines this approximation by extending the vector until the approximation is precise.

Whereas the second characterization directly defines the states of a hypothesis automaton, each occurring output vector corresponds to one state in the hypothesis automaton, the spanning tree on $\mathcal{L}p$ is used to determine the corresponding transitions.

In order to characterize the relation between hypothesis models and a corresponding SUL let $\mathcal{M} = \langle S, s_0, \Sigma, \Omega, \delta, \lambda \rangle$ and $\mathcal{M}' = \langle S', s_0', \Sigma, \Omega, \delta', \lambda' \rangle$ be two Mealy machines with shared alphabets and $\mathcal{D}$ be a set of words in Σ^*. Then we call a surjective function $f_{\mathcal{D}} : S \to S'$ *existential $\mathcal{D}$-epimorphism* between $\mathcal{M}$ and $\mathcal{M}'$, if for all $s' \in S'$ there exists an $s \in S$ with $f_{\mathcal{D}}(s) = s'$ such that for all $\alpha \in \Sigma$ and all $d \in \mathcal{D}$: $f_{\mathcal{D}}(\delta(s, \alpha)) = \delta'(s', \alpha)$ and $\lambda^*(s, d) = \lambda'^*(s', d)$.

Please note that active learning conceptually deals with the canonical Mealy machine $\mathcal{C}([\![SUL]\!])$ for a given SUL, and not with the perhaps much larger Mealing machine of the SUL itself. This reflects the fact that it is not possible to observe the difference of these two Mealy machines.

Exploiting the fact that all the algorithms considered in the following maintain a successively growing extended spanning tree for the arising hypothesis automaton $H = \langle S_H, h_0, \Sigma, \Omega, \delta_H, \lambda_H \rangle$, i.e., a prefix-closed set of word reaching all its states and covering all transitions, it is quite straightforward to establish that all these hypothesis models are images of $\mathcal{C}([\![SUL]\!])$ under a canonical existential $\mathcal{D}$-epimorphism, where $\mathcal{D}$ is the set of distinctive futures underlying the hypothesis construction.

- define $f_{\mathcal{D}} : S_{SUL} \to S_H$ by $f_{\mathcal{D}}(s) = h$ in the following fashion: if there exists a $w \in \mathcal{S}p \cup \mathcal{L}p$ with $\delta(s_0, w) = s$, then $h = \delta_H(h_0, w)$. Otherwise h may be chosen arbitrarily.
- In order to establish the defining properties of $f_{\mathcal{D}}$, it suffices to consider the states reached by words in the spanning tree. For all the considered hypothesis construction algorithms this straightforwardly yields:
  - $f_{\mathcal{D}}(\delta(s, \alpha)) = \delta_H(h, \alpha)$ for all $\alpha \in \Sigma$, which reflects the characterization from below.
  - $\lambda^*(s, d) = \lambda^*_H(h, d)$ for all $d \in \mathcal{D}$, which follows from the maintained characterization from above.

This also shows that canonical, existential $\mathcal{D}$-epimorphisms quite faithfully reflect all the knowledge maintained during active learning.

However, please note the difference between the k-epimorphisms introduced in the previous section and (canonical) existential D-epimorphisms considered here:

- whereas the difference of k and $\mathcal{D}$ is not crucial, as one could have used also D-epimorphisms in the previous sections, with $\mathcal{D} = \Sigma^k$ instead of k. However, it is important for complexity reasons. Black box systems do not support the polynomial time inductive construction of k-distinguishability. Rather they require the explicit construction of distinguishing futures. We will see that it is possible to limit the size of $\mathcal{D}$ to the index of $\equiv_{[\![SUL]\!]}$.
- the role of "existential" is crucial: it reflects the fact that $f_{\mathcal{D}}$ must deal with unknown states, i.e., state that not yet have been encountered. Thus the characterization can only be valid for the already encountered states.

Most active learning algorithms, and all the variants we are considering in the following, can be proven correct using a variant of the three-step proof pattern introduced in the previous section:

- **Invariance:** The number of states of each hypothesis automaton never exceeds the index of $\equiv_{[\![SUL]\!]}$. This follows from the fact that only distinguishable states are split (cf. definition of canonical Mealy machines).

– **Progress:** Before the final partition is reached, it is guaranteed that the Equivalence Oracle provides a counterexample, i.e., an input word which leads to a different output on the SUL and on the hypothesis. As the algorithms maintain a spanning tree for the states of H, this difference can only be resolved by splitting at least one state, thus increasing the state count.
– **Termination:** The partition refinement process terminates after at most index of $\equiv_{[\![SUL]\!]}$ many steps. This is a direct consequence of the just described properties invariance and progress.

4 First Variant: Direct Hypothesis Construction

The DHC (Direct Hypothesis Construction) algorithm, whose kernel is outlined in Algorithm 2, follows the idea of a breadth-first search for states for an automaton being constructed on-the-fly. It is particularly intuitive, as each step can be followed on the (intermediate) hypothesis models which, at any time, visualizes the state of knowledge.

The algorithm uses a queue of states to be explored, which is initialized with the states of the spanning tree to be maintained (line 2 in Algorithm 2). Explored states are removed from the queue, while the successors of discovered, provably new states (states with a new extended output signature which is defined by not only comprising one step futures, but also the increasing set of distinguishing futures produced by the algorithm) are enqueued (line 14 in Algorithm 2).

Algorithm 2. Hypothesis construction of the DHC algorithm

```
Input: A set of access sequences $\mathcal{S}p$, a set of suffixes $\mathcal{D}$, a set of inputs $\Sigma$
Output: A Mealy machine $\mathcal{H} = \langle S, s_0, \Sigma, \Omega, \delta, \lambda \rangle$
 1: create states in hypothesis $\mathcal{H}$ for all access sequences
 2: add states of $\mathcal{H}$ into queue $Q$
 3: while $Q$ is not empty do
 4:    $s :=$ dequeued state from $Q$
 5:    $u :=$ access sequence to $s$
 6:    for $d \in \mathcal{D}$ do
 7:       $o := \mathrm{mq}(u \cdot d)$
 8:       set $\lambda(s, d) = o$
 9:    end for
10:    if exists state $s'$ with same output signature (i.e., $\lambda$) as $s$ then
11:       reroute all transitions in $\mathcal{H}$ that lead to $s$ to $s'$
12:       remove $s$ from $\mathcal{H}$
13:    else
14:       create and enqueue successors for all inputs in $\Sigma$ of $s$ into $Q$
15:             that are not in $\mathcal{S}p$
16:    end if
17: end while
18: Remove information about $d \in \mathcal{D} \setminus \Sigma$ from $\lambda$
19: return $\mathcal{H}$
```

The DHC algorithm starts with a one-state hypothesis, which just includes the initial state, reached by the empty word and with $\mathcal{D} = \Sigma$. Then it tries to *complete* the hypothesis, which means that for every state the extended signature, i.e., its behavior under $\mathcal{D}$, is determined (lines 6-8 in Algorithm 2). States with a new extended signature are provably new states, which need to be further investigated. This is guaranteed by enqueuing all their successors (line 14 in Algorithm 2). As initially $\mathcal{D} = \Sigma$, only $1^{=}$-distinguishable states can be revealed during the first iteration. The initially one-state spanning tree is this way straightforwardly extended to comprise a prefix closed set of access sequences to all revealed states (cf. Fig. 5 and 6).

After the termination of the while loop, we easily obtain a well-formed hypothesis by eliminating from λ all symbols of $\mathcal{D}$ that are not in Σ. This step is unnecessary after the first iteration, as $\mathcal{D}$ at first only contains elements of Σ. The hypothesis automaton is then handed over to the equivalence oracle, which either signals success, in which case the learning procedure successfully terminates, or it produces a counterexample, i.e., a word $\bar{c}$ with $\lambda^*(s_0, \bar{c}) \neq \mathrm{mq}(\bar{c})$. In the latter case, $\mathcal{D}$ is enlarged by all suffixes of $\bar{c}$ (Sect. 4.1), and a new iteration of completion begins, this time starting with the just enlarged set $\mathcal{D}$ of suffixes and with all access sequences of all the states revealed in the previous iteration (the current spanning tree). As we will see, this procedure is guaranteed to eventually terminate with an hypothesis model equivalent to the SUL.

4.1 Counterexamples

In this section we address the dominant case where the current hypothesis and the SUL are not yet equivalent and the equivalence query therefore returns a counterexample, i.e., a word $\bar{c} \in \Sigma^*$ with $\lambda^*_{\mathcal{H}}(s_0, \bar{c}) \neq \mathrm{mq}(\bar{c})$. This counterexample can be used to enlarge both

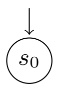
- the maintained prefix-closed set of access sequences $\mathcal{S}p$ (the spanning tree), and
- the current set of distinguishing suffixes $\mathcal{D}$

and therefore the size of the hypothesis automaton.

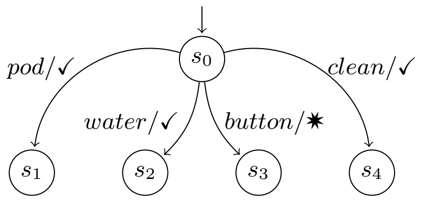
A very simple strategy has been proposed in [40] for deterministic finite automata, where simply all suffixes of a counterexample are added to $\mathcal{D}$. As we will see, this strategy is guaranteed to realize the above extension of $\mathcal{S}p$ without requiring any analysis of the counterexample, however at the prize of a fast growth of $\mathcal{D}$. A slightly improved version for Mealy machines has been proposed in [52].

In order to establish that the simple methods of [40] really works, let us introduce the following notation: for a $u \in \Sigma^*$ let $[u]_{\mathcal{H}}$ be the unique word in $\mathcal{S}p$ that reaches $\delta^*(s_0, u)$ in the hypothesis model. Then we are able to prove:

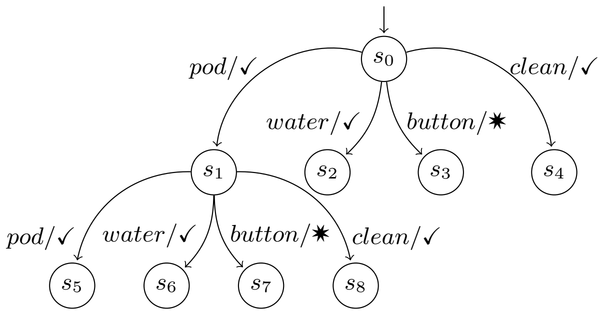
Theorem 2 (Counterexample Decomposition). *For every counterexample $\bar{c}$ there exists a decomposition $\bar{c} = u\alpha v$ into a prefix u, an action α, and a suffix v such that $mq([u]_{\mathcal{H}}\alpha v) \neq mq([u\alpha]_{\mathcal{H}}v)$.*



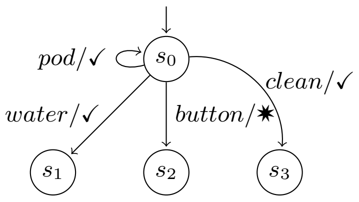
(a) incomplete starting state



(b) starting state completed



(c) state s_1 completed



(d) former state s_1 merged with starting state

Fig. 5. First steps of DHC hypothesis construction: The hypothesis at first only consists of the incomplete starting state, which is completed using membership queries. This results in new incomplete states, that are completed using additional queries. In this example the first successor of the starting state shows the same output behavior after completition, which results in this state being merged with the starting state.

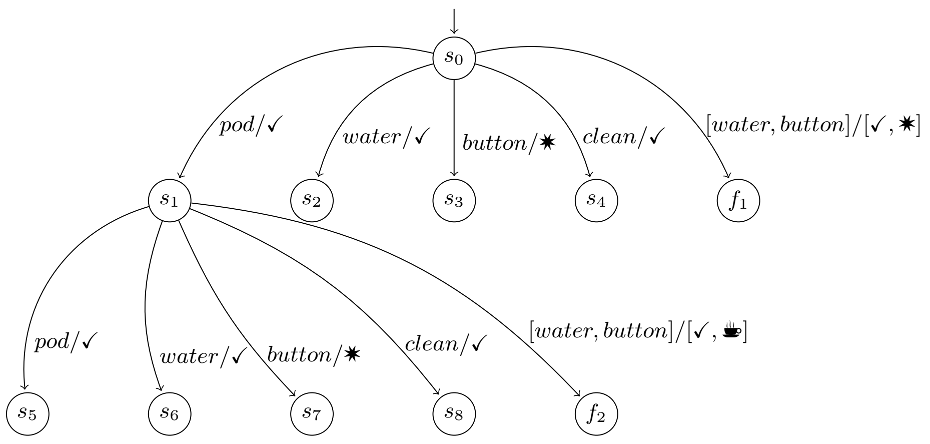


Fig. 6. An early stage of the DHC hypothesis construction, corresponding to Fig. 5(c), with the distinguishing suffix "*water button*". The diverging output for the distinguishing suffix at the states s_0 and s_1 prevents the merging of those two states, in contrast to what happens without a distinguishing suffix in Fig. 5. The states reached by distinguishing suffix transitions are named with a diverging scheme so that the state names in Fig. 5(c) are also valid in this figure for quick comparison. The letter "f" is used to express that the distinguishing suffixes reveal peeks into future behavior.

Proof. Define u as the longest prefix of $\bar{c}$ and v' as its corresponding suffix such that $\mathrm{mq}([u]_{\mathcal{H}}v') = \mathrm{mq}(\bar{c})$. As $\mathrm{mq}([\bar{c}]_{\mathcal{H}}) = \lambda^*_{\mathcal{H}}(s_0, \bar{c})$, $\bar{c}$ being a counterexample guarantees that v' has length greater than one, and that it therefore can be decomposed in αv, which concludes the proof. □

Thus adding all suffixes of counterexample $\bar{c}$ would also add v and therefore the means to separate $[u]_{\mathcal{H}}\alpha$ from all states of $\mathcal{S}p$. As a consequence, $[u]_{\mathcal{H}}\alpha$ must be added to $\mathcal{S}p$, which due to this construction even maintains the structure of a spanning tree.

More concretely, by adding v to the signature, the hypothesis construction of Algorithm 2 will automatically move $[u]_{\mathcal{H}}\alpha$ from $\mathcal{L}p$ to $\mathcal{S}p$, and add all one-letter extensions of $[u]_{\mathcal{H}}\alpha$ to the queue of to be investigated words.

At the same time, Theorem 2 also suggests that it would suffice to simply extend $\mathcal{D}$ by v. We will see in the next section that this does not only require to simply decompose the counterexamples (cf. Algorithm 4), but that there are some additional complications.

Example 6 (Analyzing a counterexample). We are learning the coffee machine from Example 2 and have produced the hypothesis in Figure 7. An equivalence query returns

$$\bar{c} \; = \; \textit{pod water pod water button}$$

as a counterexample. We have $\lambda^*_{\mathcal{H}}(q_o, \bar{c}) = ✷ \neq ☕ = \mathrm{mq}(\bar{c})$. Table 1 shows the decomposition of the counterexample into prefix, symbol, and suffix for all

Table 1. Analysis of counterexample from Example 6

Index	u	$[u]_{\mathcal{H}}$	α	v	Output
1	ϵ	ϵ	*pod*	*water pod water button*	☕
2	*pod*	ϵ	*water*	*pod water button*	☕
3	*pod water*	ϵ	*pod*	*water button*	☕
4	*pod water pod*	ϵ	*water*	*button*	✷
5	*pod water pod water*	ϵ	*button*	ϵ	✷

indices of the counterexample. The output "flips" from "☕" to "✷" between prefixes "*pod water*" and "*pod water pod*". Both prefixes have the empty word as access sequence in the hypothesis. We have

$$\mathrm{mq}([\mathit{pod\ water}]_{\mathcal{H}}\, \mathit{pod} \cdot\ \mathit{water\ button}) \neq \mathrm{mq}([\mathit{pod\ water\ pod}]_{\mathcal{H}} \cdot\ \mathit{water\ button}).$$

Adding "*water button*" to the set of suffixes will result in discovering a new state, reached by "*pod*", in the next round of hypothesis construction. The effect of adding this suffix is illustrated in Fig. 6. □

4.2 Putting It Together

We have discussed how to construct a hypothesis incrementally and how to treat counterexamples in the previous two sections. Iterating these two phases will eventually lead to a hypothesis that is behaviorally equivalent to the system under learning:

Theorem 3 (Correctness and Termination of DHC). *Iterating the DHC hypothesis construction and adding all suffixes of counterexamples to the set of suffixes $\mathcal{D}$ will lead a model that is behaviorally equivalent to the SUL with less than index of $\equiv_{[\![SUL]\!]}$ equivalence queries.*

The proof follows the three step pattern introduced in Section 2.3:

– **Invariance:** The state construction via the equivalence induced by $\mathcal{D}$ guarantees that the number of states of the hypothesis automaton can never exceed the number of states of $\mathcal{C}([\![SUL]\!])$.
  The number of states of each hypothesis automaton never exceeds the index of $\equiv_{[\![SUL]\!]}$. This follows from the fact that only distinguishable states are split (cf. definition of canonical Mealy machines).
– **Progress:** The equivalence oracle provides new counterexamples as long as the behavior of the hypothesis does not match the behavior of $\mathcal{C}([\![SUL]\!])$, and the treatment of counterexamples guarantees that at least one additional state is added to the hypothesis automaton for each counterexample.
– **Termination:** The partition refinement process terminates after at most index of $\equiv_{[\![SUL]\!]}$ many steps. This is a direct consequence of the just described properties invariance and progress.

Finally, let us consider the complexity of the DHC approach. The complexity of learning algorithms is usually measured by the number of membership resp. equivalence queries required to produce a final model. Let n denote the number of states in the final hypothesis, k the size of the set of inputs, and m the length of the longest counterexample. Then we have:

Theorem 4 (Complexity of DHC). *The DHC algorithm terminates after at most $n^3mk + n^2k^2$ membership queries and n equivalence queries.*

From Theorem 3 we know that the DHC algorithm will never require more than n equivalence queries, and therefore at most n iterations of the DHC kernel (Algorithm 2). In each of these iterations at most m new suffixes are added to $\mathcal{D}$, which is initialized with the k elements of Σ. Thus the size of $\mathcal{D}$ is bound by $k + mn$. Moreover, the number of transitions that need to be considered in an iteration never exceeds nk, which limits the number of membership queries per iteration to $O(n^2mk + nk^2)$ membership queries per round. The theorem now follows from the fact that there will never be more than n such iterations.

The DHC algorithm is a fully-functional learning algorithm for Mealy machines, which, due to its simplicity and its intuitive hypothesis construction, eases an initial understanding. Moreover, as the DHC algorithms is guaranteed to maintain a suffix closed set of distinguishing futures $\mathcal{D}$, one can prove that all intermediate hypothesis automata are guaranteed to be canonical, which means in particular that each iteration produces a new set of accepted runs.

The DHC algorithm leaves room for a number of optimizations, some of which were already covered by L^*, the first active learning algorithm. The following section describes an adaptation of L^* for Mealy machines, which, due to the L^*-specific data structure, avoids a factor n in the complexity. Additionally, following [51], we will show how also the factor m can be almost fully avoided in order to arrive at an $O(n^2k + nk^2 + n\log(m))$ algorithm. The algorithm of [51], however, no longer guarantees that the intermediate hypothesis automata are canonical. In the next section we will see that also this problem can be overcome.

5 The L^*_M Algorithm

The DHC algorithm has two major shortcomings: hypothesis automata are constructed from scratch in each round and all suffixes of each found counterexample are added to $\mathcal{D}$ before starting the next iteration. This leads to many unnecessary membership queries, which, in practice, may be rather expensive, as they may involve, e.g., communication over the network, in order to access remote resources. In this section, we present a modified L^* learning algorithm for Mealy machines that is also optimized to avoid these sources of inefficiency.

First, we introduce observation tables, the characteristic data structure of the original L^* algorithm [3], which support the incremental construction of hypothesis models and thus avoids the first source of inefficiency. The second source of inefficiency is then overcome in Section 5.2, where an optimized treatment of counterexamples is presented. Subsequently, Section 5.3 provides an estimate of the worst-case complexity of the L^*_M algorithm, as usually measured in required queries, before the algorithm is illustrated along on our running example, the coffee machine.

5.1 Observation Table

The DHC algorithm directly operates on the hypothesis model, which it reconstructs during each iteration. In this section we will present the commonly used *observation tables*, which are essentially mappings $Obs(\mathcal{U}, \mathcal{D}) : \mathcal{U} \times \mathcal{D} \rightarrow \Omega$, where $\mathcal{U} = \mathcal{S}p \cup \mathcal{L}p$ is as set of prefixes and $\mathcal{D}$ the considered set of suffixes. Observation tables represent the outcome of membership queries for words ud, with $u \in \mathcal{U}$ and $d \in \mathcal{D}$. This can be visualized in table form: rows are labeled by prefixes and columns by suffixes. The table cells contain the result of the corresponding membership query. Table 2 shows an observation table that corresponds to continuing the first steps of hypothesis construction from the example presented in Fig. 5.

In Section 4.2, we have merged states with identical signatures. In observation tables, we identify prefixes with identical rows. We write Obs_u to denote the mapping from $\mathcal{D}$ to Ω that is represented by the row labeled with u. In observation tables the part representing the *access sequences* from $\mathcal{S}p$ are collected in the upper part, whereas the not yet covered one-letter extensions ($\mathcal{L}p$), which complete the information about the transitions, are collected below.

The breadth-first search pattern we used in Section 4.2 to find new states is reflected here by establishing the *closedness* of the observation table: a table is closed if all transitions lead to already established states, i.e., if for every $u \in \mathcal{L}p$ there exists a $u' \in \mathcal{S}p$ with $Obs_u = Obs_{u'}$. Closedness is established by successively adding each $u \in \mathcal{L}p$ which does not yet have a matching state in $\mathcal{S}p$ yet to $\mathcal{S}p$, combined with adding all its one letter continuations to $\mathcal{L}p$. Please note that this directly corresponds to the enqueuing process in line 14 of the DHC algorithm.

The resulting procedure is shown in Algorithm 3. We extend $\mathcal{S}p$ and fill table cells until closedness is established on the table. As we extend $\mathcal{S}p$ only by elements of $\mathcal{L}p$, the set of access sequences will be prefix closed at any point. It represents an extended spanning tree for the hypothesis that can be constructed from the observation table.

From a closed observation table we can construct a hypothesis in a straightforward way. We construct $\mathcal{H} = \langle S, s_0, \Sigma, \Omega, \delta, \lambda \rangle$ from $Obs(\mathcal{U}, \mathcal{D})$ as follows:

– Every state in S corresponds to one word in $\mathcal{S}p$.
– the initial state s_0 will correspond to the the empty word ϵ.
– the transition function is defined by $\delta(u, \alpha) = u'$ where $u' \in \mathcal{S}p$ with $Obs_{u\alpha} = Obs_{u'}$.
– the output function is defined as $\lambda(u, \alpha) = Obs(u, \alpha)$.

It is easy to establish that this automaton is well defined: the closedness of the observation table guarantees that the transition function is well defined, and $\mathcal{D} \supseteq \Sigma$ that the output function is well defined.

Algorithm 3. Close table

```
Input: An observation table $Obs(\mathcal{U}, \mathcal{D})$
Output: A hypothesis $\mathcal{H}$
1: repeat
2:     fill table by $\mathrm{mq}(uv)$ for all pairs $u \in \mathcal{U}$ and $v \in \mathcal{D}$, where $Obs(u, v) = \varnothing$.
3:     if $\exists u \in \mathcal{L}p\ \forall u' \in \mathcal{S}p.\ Obs_u \neq Obs_{u'}$ then
4:         $\mathcal{S}p := \mathcal{S}p \cup \{u\}$
5:         $\mathcal{L}p := (\mathcal{L}p \cap \{u\}) \ \cup \ \{u\alpha \mid \alpha \in \Sigma\}$
6:     end if
7: until closedness is established
8: return hypothesis for $Obs(\mathcal{U}, \mathcal{D})$
```

Example 7 (Observation tables). We are learning the coffee machine from Example 2. Initializing $\mathcal{S}p$ as $\{\epsilon\}$ and $\mathcal{D}$ as Σ results in the observation table in Table 2. This observation table, however, is not closed since the row for the prefix "*button*" does not match the row of ϵ. Extending $\mathcal{S}p$ by "*button*" and $\mathcal{L}p$ accordingly results in the closed table shown in Table 3. From this table we can construct the hypothesis in Figure 7. □

5.2 Analyzing Counterexamples

As discussed in Section 4.1, every counterexample contains at least one suffix that, when added to $\mathcal{D}$, leads to a violation of the closedness of the observation table, and therefore to proper progress in the hypothesis construction. In the previous section, we captured this effect by simply adding all suffixes of a counterexample to $\mathcal{D}$.

This section presents an optimization of this approach which adds exactly one suffix of each counterexample to $\mathcal{D}$, following the "reduced observation table" approach from [51]. The idea is to determine the decomposition
$\bar{c} = u\alpha v$, such that $\mathrm{mq}([u]_{\mathcal{H}}\alpha v) \ \neq \ \mathrm{mq}([u\alpha]_{\mathcal{H}}v)$
guaranteed by Theorem 2 by means of binary search. See Algorithm 4 for details.

Example 8 (Binary search for suffixes). As in Example 6, let us assume that we are learning the coffee machine from Example 2 and have produced the hypothesis in Figure 7. An equivalence query returns

$$\bar{c} \ = \ \textit{pod water pod water button}$$

as a counterexample. Table 1 shows all possible decompositions of the counterexample. The decomposition for index 1 corresponds to the membership query for the original counterexample. The decomposition for index 5 corresponds to the output of hypothesis. Table 4 shows the progress of a binary search between indices 2 and 4. For the first mid-point we get the same output as for the original counterexample. Thus, we move right. For second mid-point we get an error output. We would move left in the next step. Since the upper boundary becomes smaller than the lower boundary, and since we are moving leftwards, we will return "*water button*" as new suffix. □

Algorithm 4. Process counterexample

```
Input: counterexample $\bar{c} = \bar{c}_1 \ldots \bar{c}_m$, hypothesis $\mathcal{H}$.
Output: new suffix for $\mathcal{D}$
 1: $o_{\bar{c}} := \mathrm{mq}(\bar{c})$
 2: // binary search
 3: $lower := 2, \quad upper := m - 1$
 4: loop
 5:    $mid := \lfloor (lower + upper) / 2 \rfloor$
 6:    // prepare membership query for current index
 7:    $s := \bar{c}_1 \ldots \bar{c}_{mid-1}, \quad s' := [s]_{\mathcal{H}}, \quad d := \bar{c}_{mid} \ldots \bar{c}_m$
 8:    $o_{mid} := \mathrm{mq}(s'd)$
 9:    if $o_{mid} = o_{\bar{c}}$ then
10:       $lower := mid + 1$ // same as reference output: move right
11:       if $upper < lower$ then
12:          // since $o_{mid} = o_{\bar{c}}$ and (provably) $o_{mid+1} \neq o_{\bar{c}}$
13:          return $\bar{c}_{mid+1} \ldots \bar{c}_m$
14:       end if
15:    else
16:       $upper := mid - 1$ // not same as reference output: move left
17:       if $upper < lower$ then
18:          // since $o_{mid} \neq o_{\bar{c}}$ and (provably) $o_{mid-1} = o_{\bar{c}}$
19:          return $\bar{c}_{mid} \ldots \bar{c}_m$
20:       end if
21:    end if
22: end loop
```

Simply using the suffix provided by Algorithm 4 instead of all its suffixes, or even worse, all the suffixes of the original counterexample, does already allow to remove the factor m in the estimate of $|\mathcal{D}|$. However, it comes with a defect, which led some people to doubt its correctness: intermediate hypothesis models may not be canonical. This has, e.g., been shown up when trying to use conformance testing tools to approximate the equivalence oracle: these tools typically require canonical models as input, which led them to fail on the non-canonical hypothesis. Also, minimizing these hypotheses did not seem to be a proper way out, as the minimization may, indeed, undo the achieved progress and therefore seems to lead to a dead loop.

It turns out that this is not true. Remember that the decomposition of a counterexample guaranteed in Theorem 2 and the subsequent extraction of a new state does not depend on the underlying hypothesis automaton to be canonical. Thus one can simply go ahead as usually. One should avoid to minimize the arising hypotheses, however, as this would (partially) undo the gained progress. A good heuristic is to simply reuse the same counterexample on the steadily growing non-canonical hypothesis until it fails to serve as a counterexample. Of course, canonicity is sacrificed for the intermediate hypothesis models, and can only be guaranteed for the final result.

There is, however, a way to maintain that the intermediate hypothesis models are canonical without impairing the complexity in terms of required membership and equivalence queries. The solution is based on generalizing the notion of suffix completeness. This is best understood by first considering the situation for suffix closed suffix sets in more detail:

For suffix closed $\mathcal{D}$, it is is easy to establish that for any two states u, u' from $\mathcal{S}p$ with $Obs(u, \alpha v) \neq Obs(u', \alpha v)$ we also have $Obs([u\alpha]_{\mathcal{H}}, v) \neq Obs([u'\alpha]_{\mathcal{H}}, v)$. This property can be exploited to show that the underlying Mealy machine is canonical by induction on the length of v. Please note, however, that after each step $u\alpha$ and $u'\alpha$ must be replaced by the corresponding access sequence $[u\alpha]_{\mathcal{H}}$ resp. $[u'\alpha]_{\mathcal{H}}$ in order to maintain the applicability of the induction hypothesis.

Our new notion of *semantic suffix closedness* aims at the pattern of the above-mentioned property of suffix closed suffix sets:

Definition 4 (Semantic Suffix Closedness). *Let $\mathcal{H}$ be the hypothesis model for $Obs(\mathcal{U}, \mathcal{D}) : (\mathcal{S}p \cup \mathcal{L}p) \times \mathcal{D} \to \Omega$. Then $\mathcal{D}$ is called* semantically *suffix closed for $\mathcal{H}$, if for any two states $u, u' \in \mathcal{S}p$ and any decomposition $v_1v_2 \in \mathcal{D}$ of any suffix with $Obs(u, v_1v_2) \neq Obs(u', v_1v_2)$ we also have $Obs_{[uv_1]_{\mathcal{H}}} \neq Obs_{[u'v_1]_{\mathcal{H}}}$.*

Intuitively, semantic suffix closed means that the “duty” of the suffix v_2 of v_1v_2 to split uv_1 and $u'v_1$ can be delegated to other members of $\mathcal{D}$.

It turns out that this weaker property is already sufficient to guarantee that $\mathcal{H}$ is canonical:

Theorem 5 (Semantic Suffix Closedness). *Every hypothesis constructed from an observation table with semantically suffix closed $\mathcal{D}$ is canonical.*

Theorem 5 can be proven analogously to the case of suffix closedness. One only has to change from an induction on the length of v to an induction on the sum of the lengths of all the suffixes required to cover the roles of all suffixes of v_1v_2.

We will see that this notion allows us to select “missing” suffixes which are guaranteed to enlarge the size of the hypothesis automaton without posing any membership queries. Rather, when applicable, they replace equivalence queries, and having iteratively established semantic suffix closedness, we are guaranteed to have a canonical hypothesis. In particular, this allows us to apply standard conformance testing tools to approximate the equivalence oracle.

Algorithm 5 provides a new suffix $d \in \mathcal{D}$ that leads to a proper refinement of the hypothesis $\mathcal{H}$ whenever $\mathcal{H}$ is not canonical. Thus “standard” equivalence queries need only be applied in cases when $\mathcal{H}$ is canonical.

It starts from a point of canonicity violation, i.e., two states, represented by their access sequences $u, u' \in \mathcal{S}p$, that can be distinguished by a suffix $d \in \mathcal{D}$, but which cannot be distinguished in the current topology of $\mathcal{H}$ (lines 5 and 6). As u and u' are not separated by the topology of $\mathcal{H}$, there must be a prefix of d which leads u and u' to the same state v of $\mathcal{H}$. Lines 8 to 12 determine the shortest such prefix p. Now, by definition, the suffix of d resulting from cutting of p is guaranteed to split v and therefore to refine $\mathcal{H}$. This guarantees that with each element added to d the hypothesis will grow by a least one state.

Algorithm 5. Closing canonicity defects by refinement

```
Input: An observation table $Obs(\mathcal{U}, \mathcal{D})$, a hypothesis $\mathcal{H}$
Output: A new suffix for $\mathcal{D}$ or 'ok'
 1: $P :=$ Partition on $\mathcal{S}p$, computed by Algorithm 1
 2: if $\mathcal{H}$ canonical, i.e., $|p_i| = 1$ for all $p_i \in P$ then
 3:     return 'ok'
 4: end if
 5: Let $u \neq u' \in p_i$
 6: Let $d = \alpha_1 \dots \alpha_m \in \mathcal{D}$, such that $Obs(u, d) \neq Obs(u', d)$
 7: $i := 0$
 8: while $u \neq u'$ do
 9:     $i := i + 1$
10:     $u := [u\alpha_i]_{\mathcal{H}}$
11:     $u' := [u'\alpha_i]_{\mathcal{H}}$
12: end while
13: return $\alpha_{i+1} \dots \alpha_m$
```

5.3 The Resulting Algorithm

Combining the algorithms presented in the previous sections, we construct the L^*_M learning algorithm, shown in Algorithm 6: we loop hypothesis construction and processing of counterexamples as we did already in Section 3. For hypothesis construction, we repeat closing the table and enforcing semantic suffix-closedness until the table is closed and semantically suffix-closed.

The correctness of the overall algorithm follows from the same three arguments that were used to prove Theorem 3 together with the arguments establishing the correctness of the single steps given in the this section.

Algorithm 6. L^*_M

```
Input: A set of inputs $\Sigma$
Output: A Mealy machine $\mathcal{H} = \langle S, s_0, \Sigma, \Omega, \delta, \lambda \rangle$
 1: loop
 2:     repeat
 3:         construct $\mathcal{H}$ by Algorithm *Close table*
 4:         check semantic suffix-closedness by Algorithm *Check semantic suffix-closedness*
 5:         if Algorithm *Check semantic suffix-closedness* returns new suffix $d$ then
 6:             $\mathcal{D} := \mathcal{D} \cup \{d\}$
 7:         end if
 8:     until semantic suffix-closedness is established
 9:     $\bar{c} := \mathrm{eq}(\mathcal{H})$
10:     if $\bar{c} =$ 'ok' then
11:         return $\mathcal{H}$
12:     else
13:         get suffix $d$ from $\bar{c}$ by Algorithm *Process counterexample*
14:         $\mathcal{D} := \mathcal{D} \cup \{d\}$
15:     end if
16: end loop
```

Theorem 6 (Correctness and Termination of L^*_M). *L^*_M will learn a model that is behaviorally equivalent to the system under learning with less than index of $\equiv_{[\![SUL]\!]}$ equivalence queries.*

Let us now consider the complexity of L^*_M. Let n denote the number of states in the final hypothesis, i.e., the index of $\equiv_{[\![SUL]\!]}$, k the size of the set of inputs, and m the length of the longest counterexample.

As $\mathcal{D}$ is initialized with Σ and every suffix d that is added to $\mathcal{D}$ guarantees a state size increase of at least one, the number of columns of the observation table is bounded by $n+k$. The number of rows is bounded by $kn+1$: one row for the empty word and k rows for every state. Thus the table will never have more than $n^2k + k^2n$ elements, and can therefore be filled out by means of $n^2k + k^2n$ membership queries. We also need membership queries for the processing of the at most n counterexamples arising from the at most n equivalence queries. Using binary search lets us estimate this number by $\log(m)$. One easily establishes that this comprises the repetitive treatment of counterexamples until they are fully exploited. Thus, altogether, we have:

Theorem 7 (Complexity of L^*_M). *The L^*_M algorithm terminates after at most $n^2k + k^2n + n \cdot \log(m)$ membership queries and n equivalence queries.*

Remark. Maintaining the output of the transitions separately, allows for starting with an initially empty $\mathcal{D}$ and therefore reduces the number of required membership queries to $n^2k + n \cdot \log(m)$. This optimization is rarely done, as k is often considered to be a small constant. This will change, because even in the already treated case studies we observed input alphabets with hundreds of symbols. Some of this complexity can of course be overcome by adequate abstraction (see also Section 6), and there are other powerful optimizations to reduce the data structures and the number of membership queries to maintain them. Popular is the introduction of observation packs which maintain suffix sets tailored to every state [4].

Finally, let us point out that all computation that is required on the table and the construction of hypothesis models is polynomial in the size of the final observation table.

5.4 Using L^*_M on the Coffee Machine Example

In this section we will apply the algorithm developed during the previous sections to the coffee machine. Assume that the SUL we are using to learn a model of the coffee machine equals the model from Figure 2.

We initialize the observation table by $\mathcal{S}p = \{\epsilon\}$, and $\mathcal{L}p = \mathcal{D} = \Sigma$. The resulting observation table is shown in Table 2. This table, however, is not closed. Adding "*button*" to the set of access sequences and extending $\mathcal{L}p$ accordingly will result in the table from Table 3.

This table is closed and we can construct a hypothesis from this table. The words from $\mathcal{S}p$ become the access sequences to the states of the hypothesis. The transition function will be defined according to the characterization of states

Table 2. Not yet closed observation table, first round

		$\mathcal{D}$			
		water	*pod*	*button*	*clean*
$\mathcal{S}p$	ϵ	✓	✓	✱	✓
$\mathcal{L}p$	*water*	✓	✓	✱	✓
	pod	✓	✓	✱	✓
	button	✱	✱	✱	✱
	clean	✓	✓	✱	✓

Table 3. Observation table, end of first round

		$\mathcal{D}$			
		water	*pod*	*button*	*clean*
$\mathcal{S}p$	ϵ	✓	✓	✱	✓
	button	✱	✱	✱	✱
$\mathcal{L}p$	*water*	✓	✓	✱	✓
	pod	✓	✓	✱	✓
	clean	✓	✓	✱	✓
	$button \cdot water$	✱	✱	✱	✱
	$button \cdot pod$	✱	✱	✱	✱
	$button \cdot button$	✱	✱	✱	✱
	$button \cdot clean$	✱	✱	✱	✱

in terms of the rows of the observation table. The output function will be constructed from the corresponding entries in the table for prefixes from $\mathcal{S}p$ and suffixes from Σ using membership queries. The resulting hypothesis is shown in Figure 7.

As discussed already in Example 8, an equivalence query returns

$$\bar{c} \;=\; pod\ water\ pod\ water\ button$$

as a counterexample. Table 1 shows all possible decompositions of the counterexample. The decomposition for index 1 corresponds to the membership query for the original counterexample. The decomposition for index 5 corresponds to the output of hypothesis. Table 4 shows the progress of a binary search between indices 2 and 4. For the first mid-point we get the same output as for the original counterexample. Thus, we move right. For second mid-point we get an error

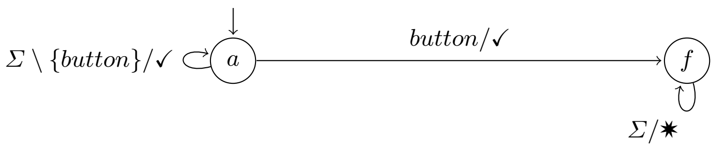


Fig. 7. $\mathcal{H}_1$ of the coffee machine

Table 4. Analysis of first counterexample

u	$[u]_{\mathcal{H}}$	lower	mid	upper	1	2	3	4	5
ϵ	ϵ				☕				
pod water pod water	ϵ								✷
pod water	ϵ	2	3	4			☕		
pod water pod	ϵ	3+1	4	4				✷	
-		4	-	4-1					

output. We would move left in the next step. Since the upper boundary becomes smaller than the lower boundary, and since we are moving leftwards, we will return "*water button*" as new suffix.

Adding "*water button*" to the set of suffixes will eventually result in the observation table in Table 5 from which we can construct the second hypothesis. The hypothesis is shown in Figure 8. Comparing the hypothesis with the minimal model for the coffee machine from Figure 3, we see that only one state is missing in the current hypothesis. The missing state is the state that is reached by the access sequence "*water*" and that could be distinguished from the initial state by the suffix "*pod button*".

Table 5. Observation table, end of second round

		$\mathcal{D}$				
		water	*pod*	*button*	*clean*	*water · button*
$\mathcal{S}p$	ϵ	✓	✓	✷	✓	✷
	button	✷	✷	✷	✷	✷
	pod	✓	✓	✷	✓	☕
	pod · water	✓	✓	☕	✓	☕
	pod · water · button	✷	✷	✷	✓	✷
$\mathcal{L}p$	*water*	✓	✓	✷	✓	✷
	clean	✓	✓	✷	✓	✷
	button · water	✷	✷	✷	✷	✷
	button · pod	✷	✷	✷	✷	✷
	button · button	✷	✷	✷	✷	✷
	button · clean	✷	✷	✷	✷	✷
	pod · pod	✓	✓	✷	✓	☕
	pod · button	✷	✷	✷	✷	✷
	pod · clean	✓	✓	✷	✓	✷
	pod · water · water	✓	✓	☕	✓	☕
	pod · water · pod	✓	✓	☕	✓	☕
	pod · water · clean	✓	✓	✷	✓	✷
	pod · water · button · water	✷	✷	✷	✷	✷
	pod · water · button · pod	✷	✷	✷	✷	✷
	pod · water · button · button	✷	✷	✷	✷	✷
	pod · water · button · clean	✓	✓	✷	✓	✷

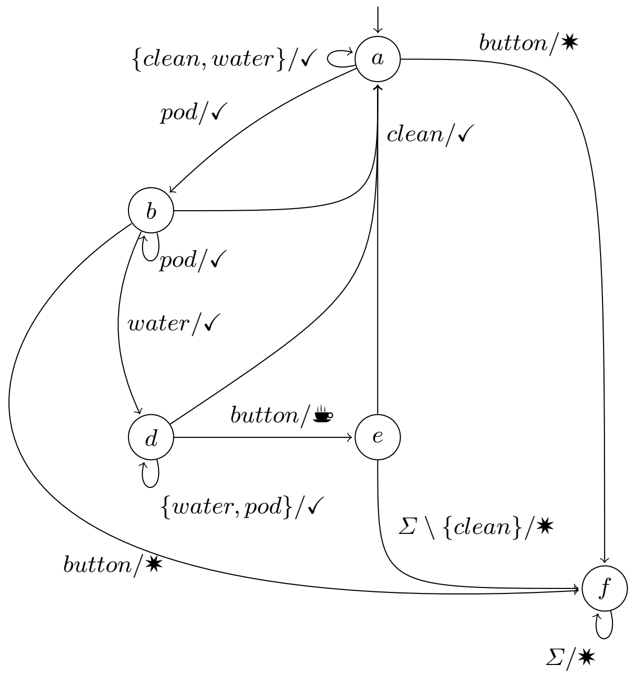


Fig. 8. $\mathcal{H}_2$ of the coffee machine

Let us assume that the second equivalence query returns

$$\bar{c} \;=\; \textit{water pod button}$$

as a counterexample. The decomposition for index 1 corresponds to the membership query for the original counterexample. The decomposition for index 3 corresponds to the output of hypothesis. Table 6 shows the (trivial) progress of a binary search between indices 2 and 2. For the first mid-point we get the same output as from the hypothesis. Thus, we move left. Since the upper boundary becomes smaller than the lower boundary, and since we are moving leftwards, we will return "*pod button*" as the second new suffix.

Using this second new suffix, we can produce an observation table from which we can construct as a hypothesis the model that is shown in Figure 3. We do not

Table 6. Analysis of second counterexample

u	$[u]_{\mathcal{H}}$	lower	mid	upper	1	2	3
ϵ	ϵ				☕		
water pod	*pod*						✷
water	ϵ	2	2	2		✷	
-		2	-	2-1			

show this table here, but leave its construction as an exercise to the reader. The next equivalence query would then return positive, indicating that we arrived at a correct model of the systems behavior.

The final table would have 25 rows and 6 columns, which could be filled by 150 membership queries. An additional $3 + 2 = 5$ membership queries were used for the processing of counterexamples. In total, we used 155 membership queries, which is much less than 258 queries, which is the estimate we get from Theorem 7. Also, we used only 3 equivalence queries instead of the worst case of 6 equivalence queries. This is typical for real systems that usually are more "talkative" than the assumed worst case from Theorem 7.

6 Challenges in Practical Applications

In the previous sections we have developed a learning algorithm for reactive input/output systems that can be modeled as Mealy machines. We have assumed a scenario in which the learning algorithm can use membership queries and equivalence queries as resources. However, in practice it will not always be obvious how to realize the required resources on an actual SUL. In this section we will briefly discuss challenges to be faced when using active learning in real-world scenarios and present common solutions and approaches to these challenges.

Whereas membership queries may often be realized via testing in practice, equivalence queries are typically unrealistic, in particular when one has to deal with black box systems.

Equivalence queries compare a learned hypothesis model with the target system for language equivalence and, in case of failure, return a counterexample exposing a difference. Their realization is rather simple in simulation scenarios: if the target system is a model, equivalence can be tested explicitly. In practice, however, the SUL will typically be some kind of black box and equivalence queries will have to be approximated using membership queries. Without assuming any extra knowledge, e.g., about the number of states of the SUL (cf. Section 2.3), such equivalence tests are in general not decidable: the possibility of having not tested extensively enough will always remain.

Model-based testing methods [14,57] have been used to simulate equivalence queries. If, e.g., an upper bound on the number of states the target system can have is known, the W-method [15] or the Wp-method [20] can be applied. Both methods have an exponential complexity (in the size of the target system and measured in the number of membership queries needed – cf. Section 2.3). The relationship between regular extrapolation and conformance testing methods is discussed in [6].

If one does not have reliable extra knowledge one can build upon, one has to resort to approximative solutions of equivalence queries, which are typically based on membership queries. In this case, conformance testing methods may not always be a wise choice. It has turned out that changing the view from "trying to proof equivalence", e.g., by using conformance testing techniques, to "finding counterexamples fast" may drastically improve performance, in particular in the early learning phases. A recent attempt to intensify research in this direction is taken by the ZULU challenge [17]. The winning solution is discussed in [28]. Key to the success here was a new approach to finding counterexamples fast, together with a new interpretation of equivalence queries as one incremental model construction rather than as a number of unrelated queries. In the ZULU scenario, which considers learning of randomly generated automata just on the basis of membership queries, this led to a surprisingly efficient realization of equivalence queries: in average, only 3 to 4 membership queries where required. It will be a major challenge to achieve similar success also in other learning scenarios.

Besides the realization of equivalence queries, which are obviously problematic in practice, there are a number of more hidden challenges which need to be resolved in a practical environment. The following paragraphs discuss such challenges according to the various characteristics of application scenarios, and illustrate that "black does not equal black" in real-life black box scenarios:

A: Interacting with real systems
The interaction with a realistic target system comes with two problems: a merely technical problem of establishing an adequate interface that allows one to apply test cases for realizing membership queries, and a conceptual problem of bridging the gap between the abstract learned model and the concrete runtime scenario.

The first problem is rather simple for systems designed for connectivity (e.g., web services or code libraries) which have a native concept of being invoked from the outside and come with documentation on how to accomplish this. Establishing connectivity may be arbitrarily complicated, however, for, e.g., some embedded systems which work within well-concealed environments and are only accessible via some proprietary GUI.

The second problem is conceptually more challenging. It concerns establishing an adequate abstraction level in terms of a communication alphabet, which on one hand leads to a useful model structure, but on the other hand also allows for an automatic back and forth translation between the abstract model level and the concrete target system level.

There is some recent work focusing on the use of abstraction in learning [1,36] and even first steps in the direction of automatic abstraction refinement [29].

B: Membership Queries
Whereas small learning experiments typically require only a few hundred membership queries, learning realistic systems may easily require several orders of magnitude more. This directly shows that the speed of the target system when processing membership queries, or as in most practical settings the corresponding test cases, is of utmost importance. In contrast to simulation environments, which typically process several thousand of queries per second, real systems may well need many seconds or sometimes even minutes per test case. In such a case, rather than parallelization, minimizing the number of required test cases is the key to success.

In [28,52] optimizations are discussed to classic learning algorithms that aim at saving membership queries in practical scenarios. Additionally, the use of filters (exploiting domain specific expert knowledge) has been proven as a practical solution to the problem [42].

C: Parameters and value domains
Active learning classically is based on abstract communication alphabets. Parameters and interpreted values are only treated to an extent expressible within the abstract alphabet. In practice, this typically is not sufficient, not even for systems as simple as communication protocols, where, e.g., increasing sequence numbers must be handled, or where authentication requires matching user/password combinations. Due to the complexity of this problem, we do not expect any comprehensive solutions here. We rather think that domain- and problem-specific approaches must be developed in order to produce dedicated solutions.

First attempts to deal with parameters range from case studies with prototypical solutions [1,53,27] to smaller extensions of the basic learning algorithms that can deal with boolean parameters [7,8]. One big future challenge will be extending active learning to models with state variables and arbitrary data parameters in a general way.

D: Reset
Active learning requires membership queries to be independent. Whereas this is no problem for simulated system, this may be quite problematic in practice. Solutions range here from reset mechanisms via homing sequences [51] or snapshots of the system state to the generation of independent fresh system scenarios. Indeed, in certain situations, executing each membership query with a separate independent user scenario may be the best one can do. Besides the overhead of establishing these scenarios, this also requires an adequate aggregation of the query results. E.g., the different user password combinations of the various used scenarios must be abstractly identified.

Due to the above problems and requirements, active learning, in practice, is inherently neither correct nor complete, e.g., due to the lack of equivalence queries. However, there does not seem to be a good alternative for dealing with black-box systems, and there are some very promising experiences: already in the project reported in [23], where the learned models had only very few states, these models helped to reorganize the corresponding test suites in order to allow a much improved test selection. In this scenario it did not harm that learned models were neither correct nor complete. They revealed parts that were ignored by the existing test suites.

In the meantime, learning technology has much improved, and we are confident to be able to extrapolate high quality behavioral models for specific application scenarios, like in the case of the Connect project [35], which focuses on connectors and protocols, i.e., on systems, where domain-specific information can be used to support regular extrapolation. In this application domain we do not expect any scalability problems, as we are in the meantime able to learn systems of tens of thousands of states and millions of transitions [49].

7 The LearnLib Framework

LearnLib [50,43] is a framework for automata learning, which includes implementations for many algorithms related to automata learning, including those presented in this chapter.

The foundation of LearnLib is an extensive Java framework of data structures and utilities, based on a set of interface agreements extensively covering concerns of active learning from constructing alphabets to tethering target systems. This supports the development of new learning components with little boilerplate code.

The component model of the LearnLib extends into the core of the learning algorithms, enabling application-fit tailoring of learning algorithms, at design- as well as at runtime. In particular, it is unique in

– comprising features for addressing real-world or legacy systems, like instrumentation, abstraction, and resetting,
– resolving abstraction-based non-determinism by alphabet abstraction refinement, which would otherwise lead to the failure of learning attempts [29],
– supporting execution and systematic experimentation and evaluation, even including remote learning and evaluation components, and, most notably, in
– its high-level modeling approach described in the next section.

7.1 Modeling Learning Solutions

LearnLib Studio, which is based on jABC [55], our service-oriented framework for the modeling, development, and execution of complex applications and processes, is LearnLib's graphical interface for designing and executing learning and experimentation setups.

A complete learning solution is usually composed of several components, some of which are optional: learning algorithms for various model types, system adapters, query filters and caches, model exporters, statistical probes, abstraction providers, handlers for counterexamples etc.. Many of these components are reusable in nature. LearnLib makes them available as easy-to-use building blocks for the graphical composition of application-fit learning experiments.

Figure 9 illustrates the graphical modeling style typical for LearnLib Studio along a very basic learning scenario. One easily identifies a common three phase pattern recurring in most learning solutions: The learning process starts with a configuration phase, where in particular the considered alphabet and the system connector are selected, before the learner itself is created and started. The subsequent central learning phase is characterized by the L^*-typical iterations, which organize the test-based interrogation of the SUL. As described in detail in the previous sections, these iterations are structured in phases of exploration, which end with the construction of a hypothesis automaton, and the (approximate) realization of the so-called equivalence query, which in practice searches for counterexamples separating the hypothesis automaton from the SUL. If this search is successful, a new phase of exploration is started in order to take care

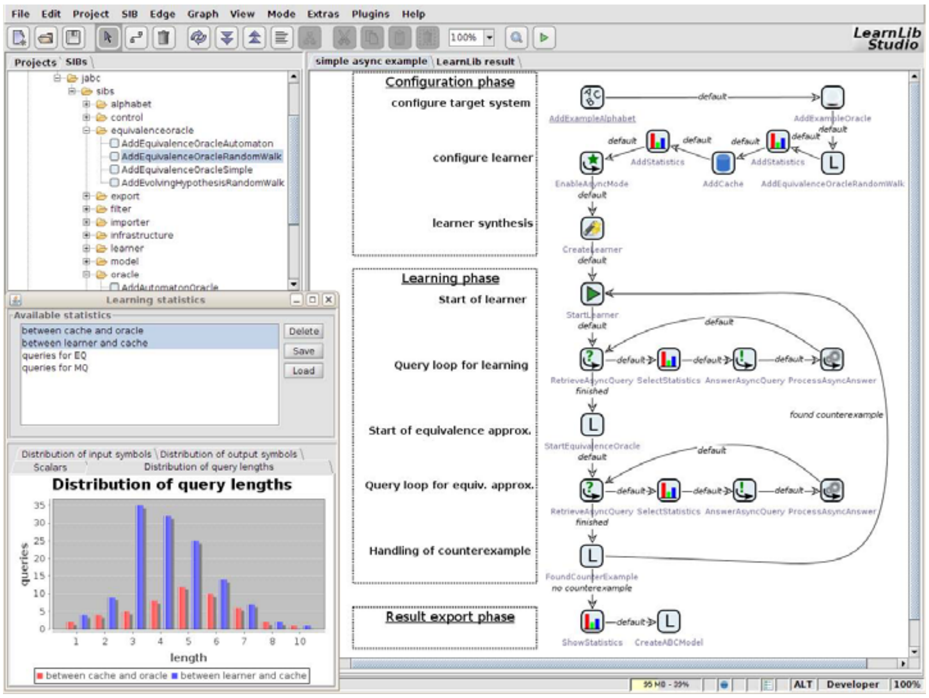


Fig. 9. Executable model of a simple learning experiment in LearnLib Studio

of all the consequences implied by the counterexample. Otherwise the learning process terminates after some postprocessing in the third phase, e.g., to produce statistical data.

Most learning experiments follow this pattern, usually enriched by application-specific refinements. Our graphical modeling environment is designed for developing such kinds of refinements by supporting, e.g., component reuse, versioning, optimization and evaluation.

8 Conclusions

In this chapter we have given an introduction to active learning of Mealy machines, an automata model particularly suited for modeling the behavior of realistic reactive systems. We have tried to build on the readers intuition by establishing links to classical automata theory, in particular concerning the minimization of finite automata based on Myhill/Nerode's famous theorem [44]. We have also discussed practical concerns, most importantly the concept of an *equivalence query* classical active learning depends upon, but also a number of other issues arising when trying to put learning technology into practice. All these considerations lead into the direction of model-based testing [56,57], where accessibility of a system, system reset, and conformance are major concerns. In fact, model-based testing provides perhaps the best practical solution to the realization of equivalence queries. In this light automata learning can be seen as a method to overcome the central hurdle of model-based testing, the availability of a model: being able to aggregate testing knowledge in an optimal fashion in terms of models enables "model-based testing without requiring models".

9 Bibliographic Notes and Further Reading

Bibliographic notes: In this chapter we have presented some basic results for Mealy machines along with an efficient learning algorithm for Mealy machines. We have tried to follow in presentation the style that is usually used in introductions to automata theory (cf. [26]). Especially Theorem 1 is a one-to-one adaptation of the Myhill/Nerode theorem [44].

For the learning algorithms, these follow the general pattern introduced by Dana Angluin [3] for deterministic finite automata. We have presented a straightforward adaption of this algorithm to Mealy machines in [41,45].

The simple pattern for handling counterexamples presented in Section 4.1 has been introduced for DFA in [40]. A slightly improved version for Mealy machines has been presented in [52]. The binary search for counterexamples was presented in [51] for DFA.

In Section 5.2, we have introduced the concept of *semantic suffix-closedness*. Our definition of semantic suffix-closedness in some respect is similar to the concept of *consistency*, introduced in [3]. While consistency is used to extend the set of suffixes by longer words, we use semantic suffix-closedness to extend the set of suffixes by shorter words. Intuitively, we propagate information in forward direction along transitions, while in Angluin's L^* it is propagated in backward direction.

Further reading: Automata learning has grown to be a wide research area in the past decades. Automata are widely used to represent knowledge gathered by learning methods. Only one branch of the field is concerned with active learning by queries (i.e., questions that the learner can ask some "teacher"). A wider perspective on the field of automata learning is given in [38,18].

The particular queries we use are *membership queries*, which correspond to a single test run on a SUL, and *equivalence queries* that compare a current hypothesis model with the actual system. The first algorithm for this scenario (called L^*) is due to Dana Angluin [3]. Using the underlying concept of query learning a number of optimizations and akin algorithms have been proposed [51,38], [4] gives a unifying overview.

We proved the practical relevance of automata learning in the context of the documentation and verification of telecommunication systems [24,23]. To meet the requirements in practical scenarios, we transferred automata learning to Mealy machines [41]. Mealy machines are widely used models of deterministic reactive systems and the development of new learning algorithms for Mealy machines is still an active field of research [52,50]. In fact, Mealy machine learning seems to dominate for practical and larger-scale applications. Examples are the learning of behavioral models for Web Services [48], communication protocol entities [11], or software components [53,49].

Recent extensions to inference methods focus on capturing further phenomena that occur in real systems. On the basis of inference algorithms for Mealy machines, inference algorithms for I/O-automata [2], timed automata [22], Petri Nets [19], and Message Sequence Charts [12,13] have been developed. With the I/O-automata model, the wide range of systems that comprise quiescence is made accessible for query learning. Timed automata model explicitly time dependent behavior. With Petri Nets, systems with explicit parallelism and distributed states are addressed.

First extensions that use complex interface alphabets with data parameters are presented in [7,54]. In [8] active learning is applied to systems with complex actions with parameters over infinite domains comprising an infinite state space.

A key enabler for dealing with infinite parameter domains and real systems is abstraction, which, however, usually is also the cause of a major problem: the introduction of non-determinism. In [29], we introduce a method for refining a given abstraction on the inputs to automatically regain a deterministic behavior on-the-fly during the learning process. Thus the control over abstraction becomes part of the learning process, with the effect that detected non-determinism does not lead to failure, but to a dynamic alphabet abstraction refinement. Like automata learning itself, this method in general is neither sound nor complete, but it also enjoys similar convergence properties even for infinite systems as long as the concrete system itself behaves deterministically, as illustrated along a concrete example.

Besides the extension of learning algorithms to cover a wider range of phenomena, the application of active learning in model checking (especially in assume-guarantee-style compositional verification) is an active field of research [16,46,39,47]. The moderate style of exploration that is used in learning is used here to ease the problem of state space explosion.

10 Exercises

1. The coffee machine presented in Example 1 has an error state that cannot be overcome by conventional operations on the machine. The manufacturer's support hotline, however, informs that it is possible to reset the machine into a working state by removing all expendables, disconnecting the machine from the power grid, waiting several minutes, and then restoring power to the machine. This procedure is called "hardreset". How can the Mealy machine specification from Example 2 be adapted to include this operation?
2. In Section 2.3 we have given very roughly two ideas for finding canonical Mealy machines from runs without using partition refinement. Develop implementations of both approaches and relate your ideas to Theorem 1 and Proposition 1.
3. Algorithm 1 computes a canonical Mealy machine for an arbitrary Mealy machine. It is based on partition refinement. One could argue that it implicitly uses distinguishing suffixes. Make this usage explicit by extending

the algorithm to construct a set of distinguishing suffixes while refining the partition on the set of states. Can you keep the size of the suffix set below n?
4. Complete the construction of the hypothesis begun in Fig. 5. What does the result look like?
5. Are there other counterexamples for the hypothesis in Fig. 7 than the one used in Example 6? If so, repeat the analysis done in Example 6 with the counterexample you discovered.
6. The manufacturer of the coffee machine has issued an updated product that can detect when a "clean" operation has been performed. Now, when being in the error state, the machine will return to the initial state when performing the "clean" procedure. What updates to the tables in Section 5.4 are necessary?
7. Elaborate the proof sketch for Theorem 5. Is semantic suffix closedness also necessary for the canonicity of a corresponding hypothesis? Try to prove or disprove.
8. Elaborate the proof sketch for the correctness of Algorithm 5.
9. Algorithm 5 finds a discriminating suffix without querying the SUL whenever the hypothesis is not canonical. Give an example of a learning process in which Algorithm 5 actually leads to an extension of the set of suffixes, and therefore to a refinement of the hypothesis automaton.

References

1. Aarts, F., Jonsson, B., Uijen, J.: Generating Models of Infinite-State Communication Protocols Using Regular Inference with Abstraction. In: Petrenko, A., Simão, A., Maldonado, J.C. (eds.) ICTSS 2010. LNCS, vol. 6435, pp. 188–204. Springer, Heidelberg (2010)
2. Aarts, F., Vaandrager, F.: Learning I/O Automata. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010. LNCS, vol. 6269, pp. 71–85. Springer, Heidelberg (2010)
3. Angluin, D.: Learning Regular Sets from Queries and Counterexamples. Information and Computation 75(2), 87–106 (1987)
4. Balcázar, J.L., Díaz, J., Gavaldà, R.: Algorithms for Learning Finite Automata from Queries: A Unified View. In: Advances in Algorithms, Languages, and Complexity, pp. 53–72 (1997)
5. Bennaceur, A., Blair, G.S., Chauvel, F., Georgantas, N., Grace, P., Howar, F., Inverardi, P., Issarny, V., Paolucci, M., Pathak, A., Spalazzese, R., Steffen, B., Souville, B.: Towards an Architecture for Runtime Interoperability. In: Margaria, T., Steffen, B. (eds.) ISoLA 2010. LNCS, vol. 6416, pp. 206–220. Springer, Heidelberg (2010)
6. Berg, T., Grinchtein, O., Jonsson, B., Leucker, M., Raffelt, H., Steffen, B.: On the correspondence between conformance testing and regular inference. In: Cerioli, M. (ed.) FASE 2005. LNCS, vol. 3442, pp. 175–189. Springer, Heidelberg (2005)
7. Berg, T., Jonsson, B., Raffelt, H.: Regular inference for state machines with parameters. In: Baresi, L., Heckel, R. (eds.) FASE 2006. LNCS, vol. 3922, pp. 107–121. Springer, Heidelberg (2006)

8. Berg, T., Jonsson, B., Raffelt, H.: Regular inference for state machines using domains with equality tests. In: Fiadeiro, J.L., Inverardi, P. (eds.) FASE 2008. LNCS, vol. 4961, pp. 317–331. Springer, Heidelberg (2008)
9. Bertolino, A., Calabro, A., Di Giandomenico, F., Nostro, N.: Dependability and Performance Assessment of Dynamic CONNECTed Systems. Formal Methods for Eternal Networked Software Systems. Springer, Heidelberg (2011)
10. Blair, G.S., Paolucci, M., Grace, P., Georgantas, N.: Interoperability in Complex Distributed Systems. Formal Methods for Eternal Networked Software Systems. Springer, Heidelberg (2011)
11. Bohlin, T., Jonsson, B.: Regular Inference for Communication Protocol Entities. Technical report, Department of Information Technology, Uppsala University, Schweden (2009)
12. Bollig, B., Katoen, J.-P., Kern, C., Leucker, M.: Replaying play in and play out: Synthesis of design models from scenarios by learning. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 435–450. Springer, Heidelberg (2007)
13. Bollig, B., Katoen, J.-P., Kern, C., Leucker, M.: *smyle*: A tool for synthesizing distributed models from scenarios by learning. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 162–166. Springer, Heidelberg (2008)
14. Broy, M., Jonsson, B., Katoen, J.-P., Leucker, M., Pretschner, A.: Model-Based Testing of Reactive Systems. LNCS, vol. 3472. Springer, Heidelberg (2005)
15. Chow, T.S.: Testing Software Design Modeled by Finite-State Machines. IEEE Trans. on Software Engineering 4(3), 178–187 (1978)
16. Cobleigh, J.M., Giannakopoulou, D., Păsăreanu, C.S.: Learning assumptions for compositional verification. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 331–346. Springer, Heidelberg (2003)
17. Combe, D., de la Higuera, C., Janodet, J.-C.: Zulu: an Interactive Learning Competition. In: Yli-Jyrä, A., Kornai, A., Sakarovitch, J., Watson, B. (eds.) FSMNLP 2009. LNCS, vol. 6062, Springer, Heidelberg (2010)
18. de la Higuera, C.: Grammatical Inference: Learning Automata and Grammars. Cambridge University Press, New York (2010)
19. Esparza, J., Leucker, M., Schlund, M.: Learning Workflow Petri Nets. In: Lilius, J., Penczek, W. (eds.) PETRI NETS 2010. LNCS, vol. 6128, pp. 206–225. Springer, Heidelberg (2010)
20. Fujiwara, S., von Bochmann, G., Khendek, F., Amalou, M., Ghedamsi, A.: Test Selection Based on Finite State Models. IEEE Trans. on Software Engineering 17(6), 591–603 (1991)
21. Grace, P., Georgantas, N., Bennaceur, A., Blair, G., Chauvel, F., Issarny, V., Paolucci, M., Saadi, R., Souville, B., Sykes, D.: The CONNECT Architecture. Formal Methods for Eternal Networked Software Systems. Springer, Heidelberg (2011)
22. Grinchtein, O., Jonsson, B., Pettersson, P.: Inference of event-recording automata using timed decision trees. In: Baier, C., Hermanns, H. (eds.) CONCUR 2006. LNCS, vol. 4137, pp. 435–449. Springer, Heidelberg (2006)
23. Hagerer, A., Hungar, H., Niese, O., Steffen, B.: Model generation by moderated regular extrapolation. LNCS, pp. 80–95. Springer, Heidelberg (2002)
24. Hagerer, A., Margaria, T., Niese, O., Steffen, B., Brune, G., Ide, H.-D.: Efficient regression testing of CTI-systems: Testing a complex call-center solution. Annual Review of Communication, Int. Engineering Consortium (IEC) 55, 1033–1040 (2001)
25. Hopcroft, J.E.: An n log n algorithm for minimizing states in a finite automaton. Technical report, Stanford, CA, USA (1971)

26. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to automata theory, languages, and computation, 2nd edn. Addison-Wesley series in computer science. Addison-Wesley-Longman, Amsterdam (2001)
27. Howar, F., Jonsson, B., Merten, M., Steffen, B., Cassel, S.: On Handling Data in Automata Learning. In: Margaria, T., Steffen, B. (eds.) ISoLA 2010. LNCS, vol. 6416, pp. 221–235. Springer, Heidelberg (2010)
28. Howar, F., Steffen, B., Merten, M.: From ZULU to RERS - Lessons Learned in the ZULU Challenge. In: Margaria, T., Steffen, B. (eds.) ISoLA 2010. LNCS, vol. 6415, pp. 687–704. Springer, Heidelberg (2010)
29. Howar, F., Steffen, B., Merten, M.: Automata learning with automated alphabet abstraction refinement. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 263–277. Springer, Heidelberg (2011)
30. Howar, F., Steffen, B., Merten, M., Margaria, T.: Practical Aspects of Active Learning. FMICS Handbook on Industrial Critical Systems. Wiley, Chichester (to appear, 2011)
31. Hungar, H., Niese, O., Steffen, B.: Domain-specific optimization in automata learning. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 315–327. Springer, Heidelberg (2003)
32. Hungar, H., Steffen, B.: Behavior-based model construction. Int. J. Softw. Tools Technol. Transf. 6(1), 4–14 (2004)
33. Inverardi, P., Spalazzese, R., Tivoli, M.: Application-layer Connector Synthesis. Formal Methods for Eternal Networked Software Systems. Springer, Heidelberg (2011)
34. Issarny, V., Bennaceur, A., Bromberg, Y.-D.: Middleware-layer Connector Synthesis. Formal Methods for Eternal Networked Software Systems. Springer, Heidelberg (2011)
35. Issarny, V., Steffen, B., Jonsson, B., Blair, G.S., Grace, P., Kwiatkowska, M.Z., Calinescu, R., Inverardi, P., Tivoli, M., Bertolino, A., Sabetta, A.: CONNECT Challenges: Towards Emergent Connectors for Eternal Networked Systems. In: ICECCS, pp. 154–161 (2009)
36. Jonsson, B.: Machine Learning and Data. Formal Methods for Eternal Networked Software Systems. Springer, Heidelberg (2011)
37. Jung, G., Margaria, T., Wagner, C., Bakera, M.: Formalizing a Methodology for Design- and Runtime Self-Healing. In: IEEE International Workshop on Engineering of Autonomic and Autonomous Systems, pp. 106–115 (2010)
38. Kearns, M.J., Vazirani, U.V.: An Introduction to Computational Learning Theory. MIT Press, Cambridge (1994)
39. Kwiatkowska, M., Norman, G., Parker, D., Qu, H.: Assume-Guarantee Verification for Probabilistic Systems. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 23–37. Springer, Heidelberg (2010)
40. Maler, O., Pnueli, A.: On the Learnability of Infinitary Regular Sets. Information and Computation 118(2), 316–326 (1995)
41. Margaria, T., Niese, O., Raffelt, H., Steffen, B.: Efficient test-based model generation for legacy reactive systems. In: HLDVT 2004: Proceedings of the Ninth IEEE International High-Level Design Validation and Test Workshop, pp. 95–100. IEEE Computer Society, Washington, DC (2004)
42. Margaria, T., Raffelt, H., Steffen, B.: Knowledge-based relevance filtering for efficient system-level test-based model generation. Innovations in Systems and Software Engineering 1(2), 147–156 (2005)

43. Merten, M., Steffen, B., Howar, F., Margaria, T.: Next Generation LearnLib. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 220–223. Springer, Heidelberg (2011)
44. Nerode, A.: Linear Automaton Transformations. Proceedings of the American Mathematical Society 9(4), 541–544 (1958)
45. Niese, O.: An Integrated Approach to Testing Complex Systems. PhD thesis, University of Dortmund, Germany (2003)
46. Pasareanu, C.S., Giannakopoulou, D., Bobaru, M.G., Cobleigh, J.M., Barringer, H.: Learning to divide and conquer: applying the L* algorithm to automate assume-guarantee reasoning. Formal Methods in System Design 32(3), 175–205 (2008)
47. Peled, D., Vardi, M.Y., Yannakakis, M.: Black Box Checking. In: Wu, J., Chanson, S.T., Gao, Q. (eds.) Proc. FORTE 1999, pp. 225–240. Kluwer Academic, Dordrecht (1999)
48. Raffelt, H., Margaria, T., Steffen, B., Merten, M.: Hybrid test of web applications with webtest. In: TAV-WEB 2008: Proceedings of the 2008 Workshop on Testing, Analysis, and Verification of Web Services and Applications, pp. 1–7. ACM, New York (2008)
49. Raffelt, H., Merten, M., Steffen, B., Margaria, T.: Dynamic testing via automata learning. Int. J. Softw. Tools Technol. Transf. 11(4), 307–324 (2009)
50. Raffelt, H., Steffen, B., Berg, T., Margaria, T.: LearnLib: a framework for extrapolating behavioral models. Int. J. Softw. Tools Technol. Transf. 11(5), 393–407 (2009)
51. Rivest, R.L., Schapire, R.E.: Inference of finite automata using homing sequences. Inf. Comput. 103(2), 299–347 (1993)
52. Shahbaz, M., Groz, R.: Inferring Mealy Machines. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 207–222. Springer, Heidelberg (2009)
53. Shahbaz, M., Li, K., Groz, R.: Learning and Integration of Parameterized Components Through Testing. In: Petrenko, A., Veanes, M., Tretmans, J., Grieskamp, W. (eds.) TestCom/FATES 2007. LNCS, vol. 4581, pp. 319–334. Springer, Heidelberg (2007)
54. Shahbaz, M., Li, K., Groz, R.: Learning Parameterized State Machine Model for Integration Testing. In: Proc. 31st Annual Int. Computer Software and Applications Conf., vol. 2, pp. 755–760. IEEE Computer Society, Washington, DC (2007)
55. Steffen, B., Margaria, T., Nagel, R., Jörges, S., Kubczak, C.: Model-driven development with the jABC. In: Bin, E., Ziv, A., Ur, S. (eds.) HVC 2006. LNCS, vol. 4383, pp. 92–108. Springer, Heidelberg (2007)
56. Tretmans, J.: Model Based Testing with Labelled Transition Systems. In: Hierons, R.M., Bowen, J.P., Harman, M. (eds.) FORTEST 2008. LNCS, vol. 4949, pp. 1–38. Springer, Heidelberg (2008)
57. Tretmans, J.: Testing Supported by Learning. Formal Methods for Eternal Networked Software Systems. Springer, Heidelberg (2011)

Model-Based Testing and Some Steps towards Test-Based Modelling

Jan Tretmans[1,2,⋆]

[1] Embedded Systems Institute, Eindhoven, The Netherlands
[2] Radboud University, Institute for Computing and Information Sciences, Nijmegen, The Netherlands
jan.tretmans@esi.nl

Abstract. Model-based testing is one of the promising technologies to increase the efficiency and effectiveness of software testing. In model-based testing, a model specifies the required behaviour of a system, and test cases are algorithmically generated from this model. Obtaining a valid model, however, is often difficult if the system is complex, contains legacy or third-party components, or if documentation is incomplete. Test-based modelling, also called automata learning, turns model-based testing around: it aims at automatically generating a model from test observations. This paper first gives an overview of formal, model-based testing in general, and of model-based testing for labelled transition system models in particular. Then the practice of model-based testing, the difficulty of obtaining models, and the role of learning are discussed. It is shown that model-based testing and learning are strongly related, and that learning can be fully expressed in the concepts of model-based testing. In particular, test coverage in model-based testing and precision of learned models turn out to be two sides of the same coin.

Keywords: model-based testing, test-based modelling, automata learning.

1 Introduction

Testing is an experimental way to check whether a (software) system does what it should do. Experiments, i.e., test cases, are applied to check whether the system under test (SUT) behaves as expected and prescribed in its specification and requirements documents. Systematic testing plays an important role in the demand for improved quality of systems and software. Testing, however, is often a manual and laborious process without effective automation, which makes it error-prone, time consuming, and very costly. Estimates are that testing takes 30-50% of the total software development effort. This leads to the quest for more effective and more efficient testing.

⋆ This work has been supported by the EU FP7 under grant number ICT-214755: QUASIMODO.

Model-based testing is a promising new technology that can contribute to increasing the efficiency and effectiveness of the testing process. In model-based testing, a model is the starting point for testing. This model expresses precisely and completely what the SUT should do, and should not do, and, consequently, it is a good basis for systematically generating test cases. Model-based testing makes it possible to generate an efficient set of test cases, including test oracles, completely automatically from a model of required SUT behaviour. In this way, model-based testing allows for test automation that goes well beyond the mere automatic execution of manually crafted test scripts, which is the current state of practice. And if the model is valid, i.e., it expresses the system requirements accurately, then all these algorithmically generated tests are provably valid, too.

In Sect. 2, this paper first discusses the ideas, concepts, ingredients, and requirements of model-based testing in general, both informally and in a more rigorous framework. Then Sect. 3 presents a specific theory for model-based testing called the **ioco** approach, where models are expressed as labelled transition systems, and correctness of an SUT with respect to its model is expressed with the **ioco**-implementation relation. This approach provides a well-defined foundation for model-based testing, and it has proved to be a good basis for several practical model-based test generation tools and their application. Sect. 2 and 3 are mainly based on [43], and are intended to give an overview of formal, model-based testing in general, and the **ioco** approach in particular.

Model-based testing starts with a model that is presumed to be correct and valid. Obtaining or constructing a valid model, however, may be difficult if the system is complex, if specifiers and designers do not make models, if the system includes legacy or third-party components, or if documentation is missing or incomplete. The emerging area of *automata learning* or *test-based modelling* aims at generating models automatically from observations made during testing, following a kind of black-box reverse engineering approach [4,37,23,34,6,49,8,36,1,40,39,3]. Sect. 4 starts with discussing some practical model-based testing issues and projects, and how learning plays a role therein although in an informal and ad-hoc way. The second part of Sect. 4 discusses learning in a more systematic way, in particular placing learning in the context of the model-based testing framework of Sect. 2 and the **ioco**-test theory of Sect. 3. Sect. 4 does not intend to give an overview of learning, nor does it present any learning algorithms. It does discuss learning of nondeterministic systems and it considers the consequences of dropping the equivalence requirement between learned model and teaching system, and compares this approach with the currently prevailing Angluin-style of learning [4], such as in the learning tool environment LEARNLIB [40].

Classification of Model-Based Testing. There are different kinds of model-based testing depending on the kind of models being used, the quality aspects being tested, the level of formality involved, and the degree of accessibility and observability of the system being tested. In this contribution we consider model-based testing as *formal*, *specification-based*, *active*, *black-box*, *functionality testing.*

It is *testing*, because it involves checking some properties of the SUT by systematically performing experiments on the real, executing SUT, as opposed to, e.g., formal verification, where properties are checked on the level of formal descriptions of the system. The kind of properties being checked are concerned with *functionality*, i.e., testing whether the system correctly does what it should do in terms of correct responses to given stimuli, as opposed to, e.g., performance, usability, reliability, or security properties. Such classes of properties are also referred to as quality characteristics. The testing is *active*, in the sense that the tester controls and observes the SUT in an active way by giving stimuli and triggers to the SUT, and observing its responses, as opposed to passive testing, or monitoring.

The basis and starting point for testing is the *specification*, which prescribes what the SUT should, and should not do. The specification is given in the form of some model of behaviour to which the behaviour of the SUT must conform. This model is assumed to be correct and valid: it is not itself the subject of testing or validation. Moreover, the testing is *black-box*. The SUT is seen as a black box without internal detail, which can only be accessed and observed through its external interfaces, as opposed to white-box testing, where the internal structure of the SUT, i.e., the code, is the basis for testing.

Finally, we deal with *formal testing*: the model, or specification, prescribing the desired behaviour is given in some formal language with precisely defined syntax and semantics. But formal testing involves more than just a formal specification. It also involves a formal definition of what a conforming SUT is, a well-defined algorithm for the generation of tests, and a correctness proof that the generated tests are sound and exhaustive, i.e., that they exactly test what they should test.

2 Model-Based Testing

In model-based testing there is a *system under test* (SUT), a *model* that serves as *specification*, and the question whether the behaviour of the SUT *conforms to* the behaviour expressed in the model. To check conformance *test cases* are constructed from the model through *test generation* and *selection. Test execution* and *analysis* of test results leads to a *verdict* whether the SUT indeed conforms to the model.

These ingredients of model-based testing are now discussed in general, both informally and in a more rigorous way. The next section will elaborate these for a particular model-based testing approach using labelled transition systems as models and **ioco** as notion of conformance.

System Under Test. The system to be tested is called the *system under test* (SUT). The SUT is a real, physical object, such as a piece of hardware, a computer program with all its libraries running on a particular processor, an embedded system consisting of software embedded in some physical device, or a process control system with sensors and actuators. The SUT is treated as a black

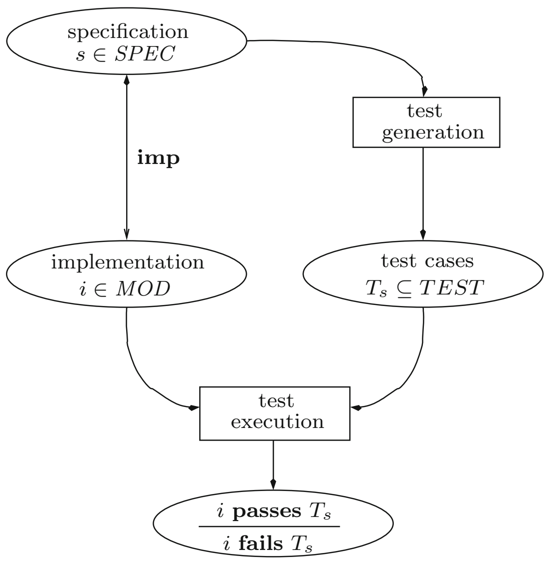


Fig. 1. The process of model-based testing

box exhibiting behaviour. A tester can only control and observe the SUT via its external interfaces, where stimuli and inputs can be provided and responses and outputs are observed. Identifying these test interfaces of the SUT, in terms of, e.g., ports, programming interfaces, message exchanges, or communication lines, is an important first step for (model-based) testing. The occurrence of input and output actions on these interfaces, together with their inter-dependencies and ordering, constitutes the behaviour of the SUT. It is this behaviour of the SUT that will be tested.

Model. The second main ingredient for model-based testing is the specification *model*. The model specifies which behaviours are allowed and which are forbidden. In our formal, model-based testing approach the model is expressed in some formal language, i.e., a language with a formal syntax and semantics. Let this language, i.e., the set of all valid expressions in this language, be denoted by $SPEC$, then a specification model s is an element of this language: $s \in SPEC$.

Conformance. The goal of model-based testing is to check whether the actual behaviour of the SUT conforms to the behaviour expressed in the model. To relate an SUT to a model, the first, static step is to map the real interfaces, inputs, and outputs of the SUT, to their abstract descriptions in the model, e.g., to map the concrete socket connection $\langle 192.168.1.1, 7890 \rangle$ to the abstract port p used in the model, and to map the concrete message with bit pattern 01010101 to the abstract message `Init`.

The second, dynamic step of relating a model to an SUT is stating precisely when an SUT correctly implements the behaviour described in a model

$s \in SPEC$. An *implementation relation*, or *conformance relation*, defines the conditions under which the behaviour of an SUT complies with the behaviour prescribed in s. Such a relation is necessary because s in itself does not completely define which SUT behaviours are correct, e.g., whether the SUT *may* or *must* implement all behaviours described in s.

An implementation relation typically answers such questions, but, if we want to define such a formal implementation relation between SUTs and specifications, we encounter a problem. Whereas a specification s is a formal object taken from the formal domain $SPEC$, an SUT is not amenable to formal reasoning. An SUT is not a formal object: it is a real, physical thing, existing in the world of material objects, on which only experiments and tests can be performed.

In order to formally reason about SUTs we do a little trick: we make the assumption that any real SUT can be modelled by some formal object i_{SUT} in a domain of models MOD. The domain MOD is a-priori chosen, may be different from $SPEC$, and is referred to as the universe of implementation models. This assumption is commonly referred to as the *test assumption* or *test hypothesis* [7,20]. Note that the test assumption presupposes a particular domain of models MOD, and that it is only assumed that a valid model i_{SUT} of the SUT exists in this domain, but not that this model i_{SUT} is a-priori known.

Thus, the test assumption allows reasoning about SUTs as if they were formal objects in MOD. This is what we will do from now on, and we call such a formal SUT model an implementation. Consequently, conformance is expressed by a formal relation between implementations and specifications, and that relation is called the *implementation relation* denoted by $\textbf{imp} \subseteq MOD \times SPEC$. An implementation $i \in MOD$ is said to be correct with respect to $s \in SPEC$ if $i \ \textbf{imp} \ s$. Implementation relations for labelled transition systems are further discussed in Sect. 3.

Test Cases. A *test case* specifies the experiment that is performed on the SUT. It specifies the inputs, or stimuli, to be supplied to the SUT, the outputs, or responses, expected from the SUT, and the ordering of these inputs and outputs. Formally, we assume a domain of test cases $TEST$ from which test cases are taken.

Test Execution and Analysis. We use the term *test execution* for applying a test case to an SUT, resulting in some observations. To execute a test case, the abstract actions of the test case must invoke the concrete interfaces of the SUT, and the concrete observations made on the SUT must be interpreted in terms of the abstract test case actions. This is called *adaptation*, and the component in the test execution environment taking care of this is usually called the *adapter*. Formally, the process of executing a test case $t \in TEST$ against an SUT is denoted by $\text{EXEC}(t, \text{SUT})$.

During test execution a number of observations is made, e.g., occurring events will be logged, or the response of the implementation to a particular stimulus will be recorded. Let there be a domain of observations OBS, then test execution, due to possible nondeterminism, leads to a set of observations: $\text{EXEC}(t, \text{SUT}) \subseteq OBS$.

Test execution $\text{EXEC}(t, \text{SUT})$ is not a formal concept but corresponds to the physical execution of a test case. This process is lifted to the level of formal models by introducing an observation function $obs : TEST \times MOD \rightarrow \mathcal{P}(OBS)$ (where $\mathcal{P}(OBS)$ denotes the power set of OBS, i.e., the set of all subsets of OBS). So, $obs(t, i_{\text{SUT}})$ is a formal expression modelling the real test execution $\text{EXEC}(t, \text{SUT})$. In the context of an observational framework consisting of $TEST$, OBS, EXEC, and obs, the test assumption can be expressed in a more precise way:

$$\forall \text{SUT} \;\; \exists i_{\text{SUT}} \in MOD \;\; \forall t \in TEST : \;\; \text{EXEC}(t, \text{SUT}) \; = \; obs(t, i_{\text{SUT}}) \tag{1}$$

This could be paraphrased as follows: for all real SUTs that we are testing, it is assumed that there is a model i_{SUT}, such that if we would put the SUT and the model i_{SUT} in black boxes and would perform all possible experiments in $TEST$ on both, then we would not be able to distinguish between the real SUT and the model i_{SUT}. Actually, this notion of testing is analogous to the ideas underlying testing equivalences [15,14].

In testing, a *verdict* is assigned based on the observations: $\nu_t : \mathcal{P}(OBS) \rightarrow \{\textbf{fail}, \textbf{pass}\}$, which allows to introduce the following abbreviation:

$$\text{SUT } \textbf{passes } t \quad \Leftrightarrow_{\text{def}} \quad \nu_t(\text{EXEC}(t, \text{SUT})) = \textbf{pass} \tag{2}$$

This is straightforwardly extended to a test suite $T \subseteq TEST$, and moreover a test suite fails if it does not pass:

$$\text{SUT } \textbf{passes } T \quad \Leftrightarrow_{\text{def}} \quad \forall t \in T : \; \text{SUT } \textbf{passes } t \tag{3}$$

$$\text{SUT } \textbf{fails } T \quad \Leftrightarrow_{\text{def}} \quad \exists t \in T : \; \text{SUT } \cancel{\textbf{passes}} \; t \tag{4}$$

Test Generation. The main gain of model-based testing is the systematic, algorithmic generation of test suites from a specification model for a given implementation relation: $gen_{\textbf{imp}} : SPEC \rightarrow \mathcal{P}(TEST)$,

The generated test cases should exactly detect those behaviours that are not correct with respect to the specification model and the implementation relation. A test suite is *sound* if all correct SUTs pass, i.e., there are no false alarms. The other way around, if no erroneous SUT passes a test suite, the test suite is called *exhaustive*, i.e., all possible failures are detected. For a specification s, a test suite T, and an implementation relation **imp**:

$$T \text{ is sound} \quad \Leftrightarrow_{\text{def}} \quad \forall i \in MOD : \; i \textbf{ imp } s \;\text{ implies }\; i \textbf{ passes } T \tag{5}$$

$$T \text{ is exhaustive} \quad \Leftrightarrow_{\text{def}} \quad \forall i \in MOD : \; i \textbf{ imp } s \quad\text{if}\quad i \textbf{ passes } T \tag{6}$$

Soundness is minimal requirement for test suites. Exhaustive test suites do not exist in practice, because detecting all potential faults in an SUT would require an infinite number of infinitely long test cases: "Program testing can be used to show the presence of bugs, but never to show their absence!" [16]. Yet, for reasoning about model-based test generation algorithms, both soundness and exhaustiveness are important concepts. A theoretically exhaustive test generation algorithm will eventually detect all possible errors if the time of testing is unbounded. Practically, this means that every error has a non-zero probability of being detected, i.e., there are no errors that are fully undetectable.

Conformance Testing. Conformance testing involves assessing, by means of testing, whether an implementation conforms to its specification. Hence, the notions of conformance, expressed by **imp**, and of test execution, expressed by **passes**, have to be linked in such a way that test execution is a (semi-) decision procedure for conformance. This can indeed be achieved if soundness and exhaustiveness are proved on models:

$$\forall i \in \mathit{MOD} : \quad i \ \mathbf{imp} \ s \quad \text{iff} \quad \forall t \in T : \ \nu_t(\mathit{obs}(t,i)) = \mathbf{pass} \tag{7}$$

Once (7) has been shown it follows that

$$\begin{array}{ll}
 & \textsc{sut}\ \mathbf{passes}\ T \\
\text{iff} & (*\ \text{definition}\ \mathbf{passes}\ T\ \ *) \\
 & \forall t \in T :\ \textsc{sut}\ \mathbf{passes}\ t \\
\text{iff} & (*\ \text{definition}\ \mathbf{passes}\ t\ \ *) \\
 & \forall t \in T :\ \nu_t(\textsc{exec}(t, \textsc{sut})) = \mathbf{pass} \\
\text{iff} & (*\ \text{test assumption (1)}\ \ *) \\
 & \forall t \in T :\ \nu_t(\mathit{obs}(t, i_{\textsc{sut}})) = \mathbf{pass} \\
\text{iff} & (*\ \text{soundness and exhaustiveness on models (7)}\ \ *) \\
 & i_{\textsc{sut}}\ \mathbf{imp}\ s \\
\text{iff} & \textsc{sut}\ \text{conforms to}\ s
\end{array}$$

Thus, testing is indeed a decision procedure for **imp**-conformance if the test assumption (1) and (7) hold. In case of test generation $\mathit{gen}_{\mathbf{imp}} : \mathit{SPEC} \rightarrow \mathcal{P}(\mathit{TEST})$ the proof obligation shall hold for any $s \in \mathit{SPEC}$:

$$\forall s \in \mathit{SPEC} \ \ \forall i \in \mathit{MOD} : \quad i \ \mathbf{imp} \ s \ \ \text{iff} \ \ i \ \mathbf{passes} \ \mathit{gen}_{\mathbf{imp}}(s) \tag{8}$$

Test Selection. A sound and exhaustive test generation algorithm can generate many more test cases than can ever be executed. Even testing the addition of two 32-bit integers, which could easily be automated by writing a test generation algorithm that enumerates all $2^{32} \times 2^{32} = 1.8\ 10^{19}$ possible test cases, would require 584,542 years of test execution if one test case would take 1 μsec.

Practical test generation algorithms use *test selection criteria* to generate a feasible and executable selection of sound, but not exhaustive test cases. The aim is to select test cases in such a way that they provide a high chance of detecting failures, and give confidence that an SUT that passes is indeed conforming, within given constraints of testing time and effort.

Selection criteria, also referred to as test adequacy criteria, are based on heuristics, experience, gut feeling, and expert domain knowledge, so human influence is prominent. General selection criteria for model-based testing express when a model is considered sufficiently *covered* by test cases. Examples are state- and transition coverage for state-based models, and condition coverage for guarded-command specifications. Selection criteria may also be specific for a particular model or domain, such as a domain expert having knowledge about particular critical behaviours.

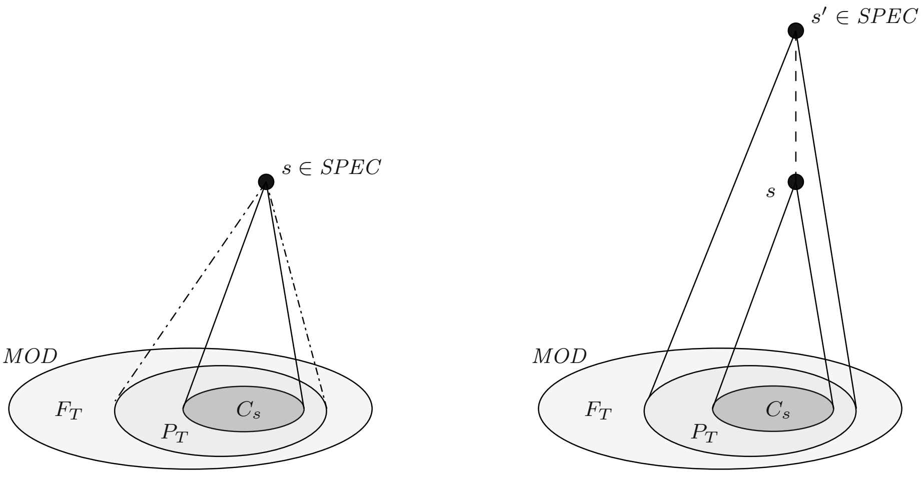


Fig. 2. Test selection as a subset of *MOD*

Test selection is an important yet difficult topic, with various approaches, see, for example, the use of test purposes [26,48], the use of metrics [13,18], approximate analysis [27], and coverage analysis [22]. We elaborate here a bit on an approach in which test selection is considered as a measure-theoretic question on the domain of implementations *MOD* [10,9]; this approach will be reconsidered in the context of learning in Sect. 4.

In Fig. 2, left-hand side, *MOD* is represented together with a specification $s \in SPEC$. The specification s partitions *MOD* into conforming implementations $C_s = \{m \mid m \ \mathbf{imp}\ s\}$ and nonconforming ones $MOD \backslash C_s$. A test suite $T \subseteq TEST$ partitions *MOD* into passing implementations $P_T = \{m \mid m \ \mathbf{passes}\ T\}$ and failing ones $F_T = MOD \backslash P_T$. Ideally, the test suite T for s is sound and exhaustive so that C_s and P_T coincide and all and only nonconforming implementations are detected by T. In practice, however, test suites are only sound, $C_s \subsetneq P_T$, so that they detect only but not all nonconforming implementations. There is an area of nonconforming yet passing implementations $P_T \backslash C_s$. Test selection aims at minimizing this area, or equivalently, optimizing the area F_T. The area F_T is a measure for the level of exhaustiveness of a sound test suite.

Suppose we have a monotonic, with $\subseteq$-increasing measure on *MOD*: $\mu : \mathcal{P}(MOD) \rightarrow \mathbb{R}_{\geq 0}$, then the coverage of a test suite T is expressed by

$$\frac{\mu(F_T)}{\mu(MOD \backslash C_s)}$$

i.e., the value of detected erroneous implementations normalized with respect to all erroneous implementations. If there is also a function $cost : \mathcal{P}(TEST) \rightarrow \mathbb{R}_{\geq 0}$ expressing the cost and effort of executing test suite T, then test selection is the optimization problem of choosing T such that the coverage is maximized and the cost is minimized. In practice, this will often amount to optimizing the coverage within given constraints on cost.

Another way of looking at the above view on test selection is depicted in the right-hand side of Fig. 2. The selected test suite T is a sound and exhaustive test suite for some other, weaker specification s', i.e., $P_T = C_{s'}$. Test selection can be cast into the specification domain as a transformation of s into s'. A sound and exhaustive test suite is then generated from s', which is a weaker specification in the sense that s' allows more conforming implementations. Suppose we can define a distance function on specifications then exhaustiveness and test selection can be quantified in the specification domain, and corresponds to minimizing the distance between s and s' constraint by maximum admissible test cost [18].

Conclusion. For reasoning about formal, model-based testing we need a formal specification language *SPEC*, a domain of models of implementations *MOD*, an implementation relation $\mathbf{imp} \subseteq \mathit{MOD} \times \mathit{SPEC}$ expressing correctness, a domain of test cases *TEST*, a test execution procedure $\mathbf{passes} \subseteq \mathit{MOD} \times \mathit{TEST}$ expressing when a model of an implementation passes a test case, a test generation algorithm $gen_{\mathbf{imp}} : \mathit{SPEC} \to \mathcal{P}(\mathit{TEST})$, a proof that a model of an implementation passes a generated test suite if and only if it is **imp**-correct, and the test assumption that any SUT can be modelled by a model in *MOD*. Then model-based testing is a decision procedure for **imp**-conformance. Yet, practical test suites are sound but not exhaustive so that test selection is necessary. Test selection quantifies the level of exhaustiveness by test coverage, and then optimizes test coverage against test cost.

The next section will elaborate most of these concepts for the formalism of labelled transition systems and the **ioco**-implementation relation. This means that we will use (variants of) labelled transition systems for *SPEC*, *MOD*, and *TEST*, that conformance is expressed as the relation **ioco** on labelled transition systems, that test execution of a labelled transition system with an implementation is defined, and that a test generation algorithm is presented that is proved to generate sound and exhaustive test suites.

Also other elaborations for other kinds of formal models are possible, e.g., Finite State Machines (FSM, Mealy Machines) [38], Abstract Data Types [20], object oriented formalisms [12], or (mathematical) functions [29].

In Sect. 4 we will use the formalizations in this section to relate model-based testing to test-based modelling, also called learning.

3 Model-Based Testing for Labelled Transition Systems

This section presents an overview of the formal test theory for labelled transition systems using the **ioco**-conformance relation; see [41,43] for a more elaborate treatment.

Models. In the **ioco**-test theory, specification models, implementations, and test cases are all expressed as labelled transition systems.

Definition 1. *A* labelled transition system with inputs and outputs *is a 5-tuple* $\langle Q, L_I, L_U, T, q_0 \rangle$ *where* Q *is a countable, non-empty set of* states*;* L_I *is a* *countable set of* input labels*;* L_U *is a countable set of* output labels*, such that* $L_I \cap L_U = \emptyset$*;* $T \subseteq Q \times (L_I \cup L_U \cup \{\tau\}) \times Q$*, with* $\tau \notin L_I \cup L_U$*, is the* transition relation*; and* $q_0 \in Q$ *is the* initial state*.*

The labels in L_I and L_U represent the inputs and outputs, respectively, of a system, i.e., the system's possible interactions with its environment. (The 'U' refers to 'uitvoer', the Dutch word for 'output', which is preferred for historical reasons, and to avoid confusion between L_O (letter 'O') and L_0 (digit zero)). Inputs are usually decorated with '?' and outputs with '!'. We use $L = L_I \cup L_U$ when we abstract from the distinction between inputs and outputs.

The execution of an action is modelled as a transition: $(q, \mu, q') \in T$ expresses that the system, when in state q, may perform action μ, and go to state q' . This is more elegantly denoted as $q \xrightarrow{\mu} q'$. Transitions can be composed: $q \xrightarrow{\mu} q' \xrightarrow{\mu'} q''$, which is written as $q \xrightarrow{\mu\cdot\mu'} q''$.

Internal transitions are labelled by the special action τ ($\tau \notin L$), which is assumed to be unobservable for the system's environment. Consequently, the observable behaviour of a system is captured by the system's ability to perform sequences of observable actions. Such a sequence of observable actions, say σ, is obtained from a sequence of actions under abstraction from the internal action τ, and it is denoted by $\overset{\sigma}{\Longrightarrow}$. If, for example, $q \xrightarrow{a\cdot\tau\cdot\tau\cdot b\cdot c\cdot\tau} q'$ $(a, b, c \in L)$, then we write $q \overset{a\cdot b\cdot c}{\Longrightarrow} q'$ for the τ-abstracted sequence of observable actions. We say that q is able to perform the *trace* $a\cdot b\cdot c \in L^*$. Here, the set of all finite sequences over L is denoted by L^*, with ϵ denoting the empty sequence. If $\sigma_1, \sigma_2 \in L^*$ are finite sequences, then $\sigma_1\cdot\sigma_2$ is the concatenation of σ_1 and σ_2. Some more, standard notations and definitions are given in Definitions 2 and 3.

Definition 2. *Let* $p = \langle Q, L_I, L_U, T, q_0\rangle$ *be a labelled transition system with* $q, q' \in Q$*,* $\mu, \mu_i \in L \cup \{\tau\}$*,* $a, a_i \in L$*, and* $\sigma \in L^*$*.*

$$\begin{array}{lll}
q \xrightarrow{\mu} q' & \Leftrightarrow_{\text{def}} & (q, \mu, q') \in T \\
q \xrightarrow{\mu_1\cdot\ldots\cdot\mu_n} q' & \Leftrightarrow_{\text{def}} & \exists q_0, \ldots, q_n : \; q = q_0 \xrightarrow{\mu_1} q_1 \xrightarrow{\mu_2} \ldots \xrightarrow{\mu_n} q_n = q' \\
q \xrightarrow{\mu_1\cdot\ldots\cdot\mu_n} & \Leftrightarrow_{\text{def}} & \exists q' : \; q \xrightarrow{\mu_1\cdot\ldots\cdot\mu_n} q' \\
q \xrightarrow{\mu_1\cdot\ldots\cdot\mu_n}\!\!\!\!\!\!/ & \Leftrightarrow_{\text{def}} & \textit{not}\ \exists q' : \; q \xrightarrow{\mu_1\cdot\ldots\cdot\mu_n} q' \\
q \overset{\epsilon}{\Longrightarrow} q' & \Leftrightarrow_{\text{def}} & q = q' \;\text{ or }\; q \xrightarrow{\tau\cdot\ldots\cdot\tau} q' \\
q \overset{a}{\Longrightarrow} q' & \Leftrightarrow_{\text{def}} & \exists q_1, q_2 : \; q \overset{\epsilon}{\Longrightarrow} q_1 \xrightarrow{a} q_2 \overset{\epsilon}{\Longrightarrow} q' \\
q \overset{a_1\cdot\ldots\cdot a_n}{\Longrightarrow} q' & \Leftrightarrow_{\text{def}} & \exists q_0 \ldots q_n : \; q = q_0 \overset{a_1}{\Longrightarrow} q_1 \overset{a_2}{\Longrightarrow} \ldots \overset{a_n}{\Longrightarrow} q_n = q' \\
q \overset{\sigma}{\Longrightarrow} & \Leftrightarrow_{\text{def}} & \exists q' : \; q \overset{\sigma}{\Longrightarrow} q' \\
q \overset{\sigma}{\not\Longrightarrow} & \Leftrightarrow_{\text{def}} & \textit{not}\ \exists q' : q \overset{\sigma}{\Longrightarrow} q'
\end{array}$$

In our reasoning about labelled transition systems we will not always distinguish between a transition system and its initial state. If $p = \langle Q, L_I, L_U, T, q_0\rangle$, we will identify the process p with its initial state q_0, and, e.g., we write $p \overset{\sigma}{\Longrightarrow}$ instead of $q_0 \overset{\sigma}{\Longrightarrow}$.

Definition 3. *Let* p *be a (state of a) labelled transition system,* P *a set of states,* $A \subseteq L$ *a set of labels, and* $\sigma \in L^*$*.*

1. $\mathit{traces}(p) =_{\text{def}} \{ \sigma \in L^* \mid p \overset{\sigma}{\Longrightarrow} \}$
2. $p\,\textbf{after}\,\sigma =_{\text{def}} \{ p' \mid p \overset{\sigma}{\Longrightarrow} p' \}$
3. $P\,\textbf{after}\,\sigma =_{\text{def}} \bigcup \{ p\,\textbf{after}\,\sigma \mid p \in P \}$
4. $P\,\textbf{refuses}\,A =_{\text{def}} \exists p \in P,\ \forall \mu \in A \cup \{\tau\} :\ p \overset{\mu}{\not\longrightarrow}$

The class of labelled transition systems with inputs in L_I and outputs in L_U is denoted as $\mathcal{LTS}(L_I, L_U)$. For technical reasons we restrict this class to *strongly converging* and *image finite* systems. Strong convergence means that infinite sequences of τ-actions are not allowed to occur. Image finiteness means that the number of non-deterministically reachable states shall be finite, i.e., for any σ, $p\,\textbf{after}\,\sigma$ shall be finite.

Representing Labelled Transition Systems. To represent labelled transition systems we use either graphs (as in Fig. 3), or expressions in a process-algebraic-like language with the following syntax:

$$B \ ::= \ a\,;B \ \mid \ \mathbf{i}\,;B \ \mid \ \mathbf{\Sigma}\,\mathcal{B} \ \mid \ B\,|[\,G\,]|\,B \ \mid \ P$$

Expressions in this language are called behaviour expressions, and they define labelled transition systems following the axioms and rules given in Table 1.

Table 1. Structural operational semantics

$$\frac{}{a\,;B \xrightarrow{a} B} \qquad \frac{}{\mathbf{i}\,;B \xrightarrow{\tau} B} \qquad \frac{B \xrightarrow{\mu} B'}{\Sigma\,\mathcal{B} \xrightarrow{\mu} B'}\ B \in \mathcal{B},\ \mu \in L \cup \{\tau\}$$

$$\frac{B_1 \xrightarrow{\mu} B_1'}{B_1\,|[G]|\,B_2 \xrightarrow{\mu} B_1'\,|[G]|\,B_2} \qquad \frac{B_2 \xrightarrow{\mu} B_2'}{B_1\,|[G]|\,B_2 \xrightarrow{\mu} B_1\,|[G]|\,B_2'} \qquad \mu \in (L\cup\{\tau\})\backslash G$$

$$\frac{B_1 \xrightarrow{a} B_1',\ B_2 \xrightarrow{a} B_2'}{B_1\,|[G]|\,B_2 \xrightarrow{a} B_1'\,|[G]|\,B_2'}\ a \in G \qquad \frac{B_P \xrightarrow{\mu} B'}{P \xrightarrow{\mu} B'}\ P := B_P,\ \mu \in L\cup\{\tau\}$$

In that table, $a \in L$ is a label, B is a behaviour expression, $\mathcal{B}$ is a countable set of behaviour expressions, $G \subseteq L$ is a set of labels, and P is a *process name*, which must be linked to a named behaviour expression by a process definition of the form $P := B_P$. In addition, we use $B_1 \,\Box\, B_2$ as an abbreviation for $\mathbf{\Sigma}\{B_1, B_2\}$, **stop** to denote $\mathbf{\Sigma}\,\emptyset$, $||$ as an abbreviation for $|[\,L\,]|$, i.e., synchronization on all observable actions, and $|||$ as an abbreviation for $|[\,\emptyset\,]|$, i.e., full interleaving without synchronization.

Input-Output Transition Systems. In the **ioco**-test theory a specification is a labelled transition system in $\mathcal{LTS}(L_I, L_U)$. In order to formally reason about an SUT the assumption is made that the SUT behaves as if it were some kind of behavioural, formal model. This assumption is referred to as the test assumption or test hypothesis, and this model is called an implementation; see Sect. 2. In the **ioco**-test theory the test assumption is that an SUT behaves as if it were a labelled transition system that is always able to perform any input action, i.e., all inputs are enabled in all states. Such a system is defined as an *input-output transition system.* The class of such input-output transition systems is denoted by $\mathcal{IOTS}(L_I, L_U) \subseteq \mathcal{LTS}(L_I, L_U)$.

Definition 4. *An* input-output transition system *is a labelled transition system with inputs and outputs* $\langle Q, L_I, L_U, T, q_0 \rangle$ *where all input actions are enabled in any reachable state:* $\forall \sigma, q :\ q_0 \overset{\sigma}{\Longrightarrow} q$ implies $\forall a \in L_I :\ q \overset{a}{\Longrightarrow}$

A state of a system where no outputs or internal actions are enabled, and consequently the system is forced to wait until its environment provides an input, is called *suspended*, or *quiescent.* An observer looking at a quiescent system does not see any outputs. This particular observation of seeing nothing can itself be considered as an event, which is denoted by δ $(\delta \notin L \cup \{\tau\})$; $p \xrightarrow{\delta} p$ expresses that p allows the observation of quiescence. Also these transitions can be composed, e.g., $p \overset{\delta\cdot ?a\cdot\delta\cdot ?b\cdot !x}{\Longrightarrow}$ expresses that initially p is quiescent, i.e., does not produce outputs, but p does accept input action $?a$, after which there are again no outputs; when then input $?b$ is performed, the output $!x$ is produced. We use L_δ for $L \cup \{\delta\}$, and traces that may contain the quiescence action δ are called *suspension traces.*

Definition 5. *Let* $p = \langle Q, L_I, L_U, T, q_0 \rangle \in \mathcal{LTS}(L_I, L_U)$.

1. *A state* q *of* p *is* quiescent, *denoted by* $\delta(q)$, *if* $\forall \mu \in L_U \cup \{\tau\} :\ q \overset{\mu}{\not\longrightarrow}$
2. $p_\delta \ =_{\text{def}}\ \langle\ Q,\ L_I,\ L_U \cup \{\delta\},\ T \cup T_\delta,\ q_0\ \rangle$,
   with $T_\delta \ =_{\text{def}}\ \{\ q \xrightarrow{\delta} q \mid q \in Q,\ \delta(q)\ \}$
3. *The* suspension traces *of* p *are* $Straces(p)\ =_{\text{def}}\ \{\ \sigma \in L_\delta^* \mid p_\delta \overset{\sigma}{\Longrightarrow}\ \}$

From now on we will usually include δ-transitions in the transition relations, i.e., we consider p_δ instead of p, unless otherwise indicated. Definitions 2 and 3 also apply to transition systems with label set L_δ.

The Implementation Relation ioco. An implementation relation is intended to precisely define when an implementation is correct with respect to a specification. We use the implementation relation **ioco**, which is abbreviated from input-output conformance. Informally, an implementation $i \in \mathcal{IOTS}(L_I, L_U)$ is **ioco**-conforming to specification $s \in \mathcal{LTS}(L_I, L_U)$ if any experiment derived from s and executed on i leads to an output (including quiescence) from i that is foreseen by s. We define **ioco** as a special case of the more general class of relations $\mathbf{ioco}_{\mathcal{F}}$, where $\mathcal{F} \subseteq L_\delta^*$ is a set of suspension traces, which typically depends on the specification s.

Definition 6. *Let* q *be a state in a transition system,* Q *be a set of states,* $i \in \mathcal{IOTS}(L_I, L_U)$, $s \in \mathcal{LTS}(L_I, L_U)$, *and* $\mathcal{F} \subseteq L_\delta^*$, *then*

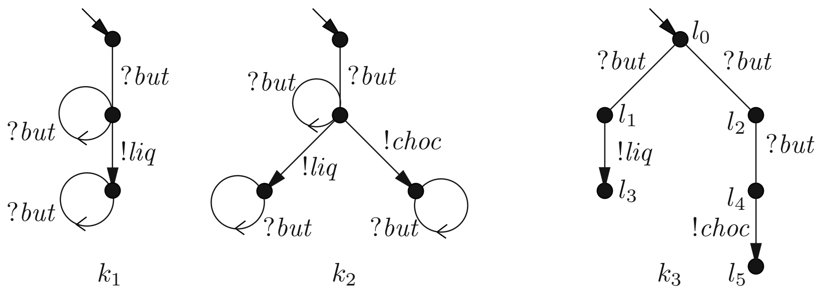


Fig. 3. Example labelled transition systems

1. $out(q) \;=_{\text{def}}\; \{\, x \in L_U \mid q \xrightarrow{x}\,\} \;\cup\; \{\, \delta \mid \delta(q)\,\}$
2. $out(Q) \;=_{\text{def}}\; \bigcup\, \{\, out(q) \mid q \in Q\,\}$
3. $i\ \mathbf{ioco}_{\mathcal{F}}\ s \quad \Leftrightarrow_{\text{def}} \quad \forall \sigma \in \mathcal{F}: \quad out(\,i\,\mathbf{after}\,\sigma\,) \;\subseteq\; out(\,s\,\mathbf{after}\,\sigma\,)$
4. $i\ \mathbf{ioco}\ s \quad \Leftrightarrow_{\text{def}} \quad i\ \mathbf{ioco}_{Straces(s)}\ s$

Example 1. Figure 3 presents three examples of labelled transition systems modelling candy machines. There is an input action for pushing a button $?but$, and there are outputs for obtaining chocolate $!choc$ and liquorice $!liq$: $L_I = \{?but\}$ and $L_U = \{!liq, !choc\}$.

Since $k_1, k_2 \in \mathcal{IOTS}(L_I, L_U)$ they can be both specifications and implementations; k_3 is not input-enabled, and can only be a specification. We have that $out(\,k_1\ \mathbf{after}\ ?but\,) = \{!liq\} \subseteq \{!liq, !choc\} = out(\,k_2\ \mathbf{after}\ ?but\,)$; so we get now k_1 **ioco** k_2, but k_2 $\mathbf{io\not{c}o}$ k_1. For k_3 we have $out(\,k_3\ \mathbf{after}\ ?but\,) = \{!liq, \delta\}$ since $\delta(l_2)$, and $out(\,k_3\ \mathbf{after}\ ?but\cdot?but\,) = \{!choc\}$, so both k_1, k_2 $\mathbf{io\not{c}o}$ k_3.

The importance of having suspension actions δ in the set $\mathcal{F}$ over which **ioco** quantifies is illustrated in Fig. 4. It holds that $out(\,r_1\ \mathbf{after}\ ?but\cdot?but\,) = out(\,r_2\ \mathbf{after}\ ?but\cdot?but\,) = \{!liq, !choc\}$, but we have $out(\,r_1\ \mathbf{after}\ ?but\cdot\delta\cdot?but\,) = \{!liq, !choc\} \supset \{!choc\} = out(\,r_2\ \mathbf{after}\ ?but\cdot\delta\cdot?but\,)$. So, without δ in these traces r_1 and r_2 would be considered implementations of each other in both directions, whereas with δ, r_2 **ioco** r_1 but r_1 $\mathbf{io\not{c}o}$ r_2.

Underspecification and the Implementation Relation uioco. The implementation relation **ioco** allows to have partial specifications. A partial specification does not specify the required behaviour of the implementation after all possible traces. This corresponds to the fact that specifications may be non-input enabled, and inclusion of *out*-sets is only required for suspension traces that explicitly occur in the specification. Traces that do not explicitly occur are called underspecified. There are different ways of dealing with underspecified traces. The relation **uioco** does it in a slightly different manner than **ioco**. For the rationale consider Example 2.

Example 2. Consider k_3 of Fig. 3 as a specification. Since k_3 is not input-enabled, it is a partial specification. For example, $?but\cdot?but\cdot?but$ is an underspecified trace, and any implementation behaviour is allowed after it. On the other hand, $?but$ is clearly specified; the allowed outputs after it are $!liq$ and δ. For the trace $?but{\cdot}?but$ the situation is less clear. According to **ioco** the expected output after $?but{\cdot}?but$ is $out(\, k_3 \,\textbf{after}\, ?but{\cdot}?but\,) = \{!choc\}$. But suppose that in the first $?but$-transition k_3 moves nondeterministically to state l_1 (the left branch) then one might argue that the second $?but$-transition is underspecified, and that, consequently, any possible behaviour is allowed in an implementation. This is exactly where **ioco** and **uioco** differ: **ioco** postulates that $?but{\cdot}?but$ is not an underspecified trace, because there exists a state where it is specified, whereas **uioco** states that $?but{\cdot}?but$ is underspecified, because there exists a state where it is underspecified.

Formally, **ioco** quantifies over $\mathcal{F} = \mathit{Straces}(s)$, which are all possible suspension traces of the specification s. The relation **uioco** quantifies over $\mathcal{F} = \mathit{Utraces}(s) \subseteq \mathit{Straces}(s)$, which are the suspension traces without the possibly underspecified traces, i.e., all suspension traces σ of s for which it is *not* possible that a prefix σ_1 of σ ($\sigma = \sigma_1{\cdot}a{\cdot}\sigma_2$) leads to a state of s where the remainder $a{\cdot}\sigma_2$ of σ is underspecified, that is, a is refused.

Definition 7. *Let $i \in \mathcal{IOTS}(L_I, L_U)$, and $s \in \mathcal{LTS}(L_I, L_U)$.*

1. $\mathit{Utraces}(s) \;=_{\mathrm{def}}\; \{\, \sigma \in \mathit{Straces}(s) \mid \forall \sigma_1, \sigma_2 \in L_\delta^*,\ a \in L_I : \sigma = \sigma_1{\cdot}a{\cdot}\sigma_2 \text{ implies not } s \,\textbf{after}\, \sigma_1 \,\textbf{refuses}\, \{a\} \,\}$
2. $i \;\textbf{uioco}\; s \;\Leftrightarrow_{\mathrm{def}}\; i \;\textbf{ioco}_{\mathit{Utraces}(s)}\; s$

Example 3. Because $\mathit{Utraces}(s) \subseteq \mathit{Straces}(s)$ it is evident that **uioco** is not stronger than **ioco**. That it is strictly weaker follows from the following example. Take k_3 in Fig. 3 as a (partial) specification, and consider r_1 and r_2 from Fig. 4 as implementations. Then $r_2 \;\cancel{\textbf{ioco}}\; k_3$ because $!liq \in out(r_2 \,\textbf{after}\, ?but{\cdot}?but\,)$ and $!liq \notin out(\, k_3 \,\textbf{after}\, ?but{\cdot}?but\,)$. But $r_2 \;\textbf{uioco}\; k_3$ because we have $?but{\cdot}?but \notin \mathit{Utraces}(k_3)$. Also $r_1 \;\cancel{\textbf{ioco}}\; k_3$, but in this case also $r_1 \;\cancel{\textbf{uioco}}\; k_3$. The reason for this is that we have $?but{\cdot}\delta{\cdot}?but \in \mathit{Utraces}(k_3)$, $!liq \in out(\, r_1 \,\textbf{after}\, ?but{\cdot}\delta{\cdot}?but\,)$ and $!liq \notin out(\, k_3 \,\textbf{after}\, ?but{\cdot}\delta{\cdot}?but\,)$.

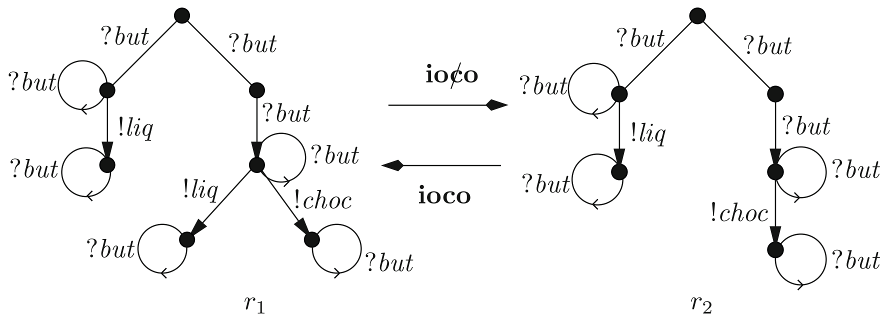


Fig. 4. More labelled transition systems

Test Cases. For the generation of test cases from labelled transition system specifications, which can test SUTs that behave as input-output transition systems, we must first define what test cases are. Then we discuss what test execution is, and what it means to pass a test. A test generation algorithm is given, and soundness and exhaustiveness are discussed.

A test case is a specification of the behaviour of a tester in an experiment carried out on an SUT. The behaviour of such a tester is also modelled as a special kind of input-output transition system, but, naturally, with inputs and outputs exchanged. Consequently, input-enabledness of a test case means that all actions in L_U (i.e., the set of outputs of the implementation) are enabled. For observing quiescence we add a special label θ to the transition systems modelling tests ($\theta \notin L$).

Definition 8. *A* test case *t for an implementation with inputs L_I and outputs L_U is an input-output transition system $\langle Q, L_U, L_I \cup \{\theta\}, T, q_0 \rangle \in \mathcal{IOTS}(L_U, L_I \cup \{\theta\})$ generated following the next fragment of the syntax for behaviour expressions, where* **pass** *and* **fail** *are process names:*

$$\begin{array}{rll} t ::= & \mathbf{pass} & \\ \mid & \mathbf{fail} & \\ \mid & \Sigma \{ \; x \; ; \; t \mid x \in L_U \cup \{a\} \; \} & \textit{for some } a \in L_I \\ \mid & \Sigma \{ \; x \; ; \; t \mid x \in L_U \cup \{\theta\} \; \} & \\ & \textit{where} \quad \mathbf{pass} := \Sigma \{ \; x \, ; \, \mathbf{pass} \; \mid \; x \in L_U \cup \{\theta\} \; \} & \\ & \qquad\quad\;\; \mathbf{fail} \;\; := \Sigma \{ \; x \, ; \, \mathbf{fail} \; \mid \; x \in L_U \cup \{\theta\} \; \} & \end{array}$$

The class of test cases for implementations with inputs L_I and outputs L_U is denoted as $\mathcal{TTS}(L_U, L_I)$. A set of test cases is called a *test suite* $T \subseteq \mathcal{TTS}(L_U, L_I)$.

Test Execution. Test cases are run by putting them in parallel with the implementation, where inputs of the test case synchronize with the outputs of the implementation, and vice versa. Basically, this can be modelled using the behaviour-expression operator $\|$. Since, however, we added the special label θ to test cases to test for quiescence, this operator has to be extended a bit, and is then denoted as $\rceil\!|$.

Because of nondeterminism in implementations, it may be the case that testing the same implementation with the same test case leads to different test runs. Test execution consists of performing all possible test runs. Each test run leads to an observation which is the trace executed until a **pass**- or **fail**-state is reached. Thus, test execution leads to a set of observations. An implementation passes a test case if and only if all its test runs lead to a pass verdict of the test case. All this is reflected in the following definition.

Definition 9. *Let $t \in \mathcal{TTS}(L_U, L_I)$ and $i \in \mathcal{IOTS}(L_I, L_U)$.*

1. Running *a test case t with an implementation i is expressed by the parallel operator $\rceil\!|$: $\mathcal{TTS}(L_U, L_I) \times \mathcal{IOTS}(L_I, L_U) \rightarrow \mathcal{LTS}(L_I \cup L_U \cup \{\delta\})$ which*

is defined by the following inference rules:

$$\frac{i \xrightarrow{\tau} i'}{t \rceil\!| \, i \xrightarrow{\tau} t \rceil\!| \, i'} \qquad \frac{t \xrightarrow{a} t', \; i \xrightarrow{a} i'}{t \rceil\!| \, i \xrightarrow{a} t' \rceil\!| \, i'} \; a \in L_I \cup L_U \qquad \frac{t \xrightarrow{\theta} t', \; i \xrightarrow{\delta}\!\!\!\!\not\;\;}{t \rceil\!| \, i \xrightarrow{\delta} t' \rceil\!| \, i}$$

2. *An* observation *of a test run of t with i is a trace of $t \rceil\!| \, i$ leading to one of the states* **pass** *or* **fail** *of t:*

$$obs(t, i) \;=_{\text{def}}\; \{ \; \sigma \in L_\delta^* \;\mid\; \exists i' : \; t \rceil\!| \, i \overset{\sigma}{\Longrightarrow} \mathbf{pass} \rceil\!| \, i' \;\text{ or }\; t \rceil\!| \, i \overset{\sigma}{\Longrightarrow} \mathbf{fail} \rceil\!| \, i' \; \}$$

3. *Implementation i* passes *test case t if all observations go to the* **pass***-state of t:*

$$i \;\mathbf{passes}\; t \quad \Leftrightarrow_{\text{def}} \quad \forall \sigma \in obs(t, i) \;\; \exists i' : \;\; t \rceil\!| \, i \overset{\sigma}{\Longrightarrow} \mathbf{pass} \rceil\!| \, i'$$

4. *An implementation i* passes *a test suite T if it passes all test cases in T:*

$$i \;\mathbf{passes}\; T \quad \Leftrightarrow_{\text{def}} \quad \forall t \in T : \; i \;\mathbf{passes}\; t$$

If i does not pass a test case or a test suite, it **fails**.

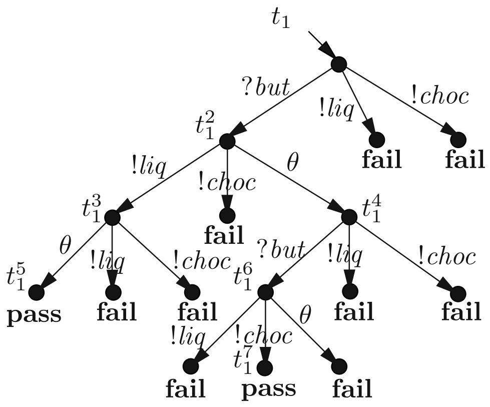


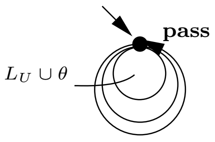
Fig. 5. A test case

Test Generation. Now all ingredients are there to present an algorithm to generate test cases from a labelled transition system specification, which test implementations for **ioco**-correctness.

Algorithm 1. *Let $s \in \mathcal{LTS}(L_I, L_U)$ be a specification, and let S be a set of states with initially $S = s$* **after** *ϵ .*

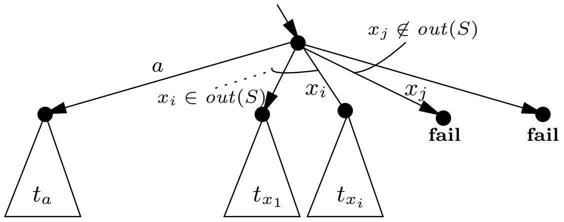
A test case $t \in \mathcal{TTS}(L_U, L_I)$ is obtained from a non-empty set of states S by a finite number of recursive applications of one of the following three nondeterministic choices:

1.



$t :=$ **pass**

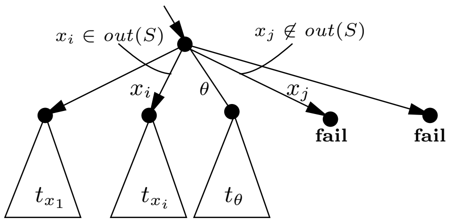
2.



$$\begin{aligned} t := \;& a \;;\; t_a \\ & \Box\, \Sigma \,\{\, x_j \;;\; \mathbf{fail} \mid x_j \in L_U,\ x_j \notin out(S) \,\} \\ & \Box\, \Sigma \,\{\, x_i \;;\; t_{x_i} \mid x_i \in L_U,\ x_i \in out(S) \,\} \end{aligned}$$

where $a \in L_I$ *such that* S **after** $a \neq \emptyset$, t_a *is obtained by recursively applying the algorithm for the set of states* S **after** a, *and for each* $x_i \in out(S)$, t_{x_i} *is obtained by recursively applying the algorithm for the set of states* S **after** x_i .

3.



$$\begin{aligned} t := \;& \Sigma \,\{\, x_j \;;\; \mathbf{fail} \mid x_j \in L_U,\ x_j \notin out(S) \,\} \\ & \Box\, \Sigma \,\{\, \theta \;;\; \mathbf{fail} \mid \delta \notin out(S) \,\} \\ & \Box\, \Sigma \,\{\, x_i \;;\; t_{x_i} \mid x_i \in L_U,\ x_i \in out(S) \,\} \\ & \Box\, \Sigma \,\{\, \theta \;;\; t_\theta \mid \delta \in out(S) \,\} \end{aligned}$$

where for each $x_i \in out(S)$, t_{x_i} *is obtained by recursively applying the algorithm for the set of states* S **after** x_i , *and* t_θ *is obtained by recursively applying the algorithm for the set of states* S **after** δ .

Algorithm 1 generates a test case from a set of states S. This set represents the set of all possible states in which the specification can be at the given stage of the test case generation. Initially $S = s\,\textbf{after}\,\epsilon = q_0\,\textbf{after}\,\epsilon$, so that the first transition of the test case is derived from the initial state(s) of the specification. Then the remaining part of the test case is recursively derived from the specification states reachable from the initial states via this first test case transition.

The algorithm is nondeterministic in the sense that in each recursive step it can be continued in three different ways. Each choice results in another, valid test case:

choice 1: The test case can be terminated by ending the recursion with the single-state test case **pass**, which is always a sound test case.

choice 2: The test case can continue with supplying an input $a \in L_I$ allowed by the specification ($S\,\textbf{after}\,a \neq \emptyset$). After action a the test case continues as t_a, which is obtained by recursive application of the algorithm with the set of states $S\,\textbf{after}\,a$. Moreover, t is prepared to accept any output of the SUT (not quiescence) that might occur before a is supplied. Analogous to t_a, each t_{x_i} is obtained from $S\,\textbf{after}\,x_i$.

choice 3: The test case can wait for an output of the SUT and check it, or conclude that the SUT is quiescent, i.e., produces 'output' δ. If the output, whether real or quiescence, is not allowed, i.e., $x_j \notin out(S)$, the test case terminates with **fail**. If the response is allowed the algorithm continues with recursively generating a test case from the set of states $S\,\textbf{after}\,x_i$.

Example 4. Test case t_1 of Figure 5 can be generated from k_3 in Figure 3 with Algorithm 1:

1. Initially, $S := k_3\,\textbf{after}\,\epsilon = \{l_0\}$.
2. In the first step input $?but$ is tried: $t_1 := ?but; t_1^2 \;\Box\; !liq; \textbf{fail} \;\Box\; !choc; \textbf{fail}$, after which $S := \{l_0\}\,\textbf{after}\,?but = \{l_1, l_2\}$.
3. The allowed outputs are checked: $out(S) = out(\{l_1, l_2\}) = \{!liq, \delta\}$. This leads to the test case $t_1^2 := !liq; t_1^3 \;\Box\; !choc; \textbf{fail} \;\Box\; \theta; t_1^4$.
4. For t_1^3 we continue with $S := \{l_1, l_2\}\,\textbf{after}\,!liq = \{l_3\}$ for which it is checked that no output occurs: $t_1^3 := !liq; \textbf{fail} \;\Box\; !choc; \textbf{fail} \;\Box\; \theta; t_1^5$.
5. The test case is stopped: $t_1^5 := \textbf{pass}$.
6. Further with t_1^4: this is the test case after quiescence has been observed; t_1^4 is generated from $S := \{l_1, l_2\}\,\textbf{after}\,\delta = \{l_2\}$. From $\{l_2\}$ another input $?but$ can be supplied: $t_1^4 := ?but; t_1^6 \;\Box\; !liq; \textbf{fail} \;\Box\; !choc; \textbf{fail}$.
7. Now output is checked: $t_1^6 := !liq; \textbf{fail} \;\Box\; !choc; t_1^7 \;\Box\; \theta; \textbf{fail}$.
8. After this the test case is stopped: $t_1^7 := \textbf{pass}$.

Soundness and Exhaustiveness. Algorithm 1 is correct, in the sense that the generated test suites are able to detect all, and only all, non-**ioco** correct implementations. This is expressed by the properties of soundness and exhaustiveness; see Sect. 2. This means that testing for **ioco** according to Algorithm 1 and Def. 9 is indeed a decision procedure for **ioco**-correctness. Of course, exhaustiveness is merely a theoretical property: for realistic systems exhaustive test suites would be infinite. But yet, exhaustiveness does express that there are no **ioco**-errors that are undetectable.

Theorem 2. *Algorithm 1 generates sound test cases, and the set of all test cases that can be generated with it is exhaustive for* **ioco** *and s.*

Example 5. In Example 4 test case t_1 of Figure 5 was generated from specification k_3 in Figure 3. For t_1 with k_1 there is one test run:
$t_1 \rceil\!| \, k_1 \xRightarrow{?but\cdot!liq\cdot\delta} \textbf{pass} \rceil\!| \, k_1'$, so k_1 **passes** t_1.
For t_1 with k_2 there are two test runs:
$t_1 \rceil\!| \, k_2 \xRightarrow{?but\cdot!liq\cdot\delta} \textbf{pass} \rceil\!| \, k_2'$, and $t_1 \rceil\!| \, k_2 \xRightarrow{?but\cdot!choc} \textbf{fail} \rceil\!| \, k_2''$, so k_2 **fails** t_1.

We had in Example 1 that $k_1, k_2 \ \mathbf{io\not{c}o} \ k_3$, so the erroneous k_2 is detected by t_1 but k_1 is not. The singleton test suite $\{t_1\}$ is indeed sound but not exhaustive.

4 From Model-Based Testing towards Test-Based Modelling

Practical Model-Based Testing. Model-based testing currently attracts a lot of interest, both from research and from companies. Academic as well as commercial model-based testing tools and services become available, and many companies are involved in trial projects [47,11,45,25,35,21]. This is triggered by the promise that model-based testing will make it possible to automate the testing process beyond the mere automation of test execution, which is the current state of practice in software testing. Once models are available, model-based testing allows automating the generation of test cases and the analysis of test results, thus making it possible to automate the complete testing process. Moreover, once models are available, also other sophisticated engineering and analysis methods are possible such as simulation, model checking, and implementation generation. Models, however, are not always easily available, because of various reasons.

Many modern systems are large, complex, distributed, dynamic, and networked systems, which are not monolithically built from scratch, but composed of components. Among these components are legacy, reused, general purpose, outsourced, third-party, and off-the-shelf components. These components are different in many aspects, such as different life cycles, different visibility and accessibility of internal details (black-box vs. white-box), and different forms of specifications and documentation, if documentation is available at all. For such components often no models are available, and because of insufficient documentation it is difficult or impossible to construct a valid model. Even for newly developed components, the construction of models typically requires specialized expertise and involves significant manual effort, in particular if the available documentation is poor or the knowledge about a system or component is concentrated in the minds of a few engineers. An additional issue is that systems evolve: components are substituted by newer versions or replaced by alternative components, components are restructured, or interaction with the system's environment changes. This adds maintainability of models as a main concern, and it may lead to models getting outdated as the system evolves.

The availability of models is therefore a key issue that inhibits the further proliferation of model-based testing and of other forms of model-based and model-driven analysis and development [42]. Even if models are available, they are often incomplete and not fully valid. This means that a straightforward model-based testing process consisting of sequentially performing the steps described in Sect. 2, Fig. 1, is too naive and does not work. Such a process, which would consist of sequentially making a model, generating test cases from the model, executing these test cases, and assigning a verdict, does not take into account that a discrepancy between actual outcomes and expected ones does not necessarily point to a fault in the SUT, but may be due to errors, misunderstanding, incompleteness, or invalidity of the model.

Consequently, practical model-based testing is not only checking an SUT with respect to a model, but also checking the model itself. Model-based testing in practice serves as a technique to detect discrepancies between the SUT and the model, but without making any judgment about which one is wrong. Only subsequent analysis and diagnosis can show whether the model shall be adapted, or the SUT shall be repaired. What we see is a process of concurrent improvement of both the SUT and the model by iteratively comparing them using tests: the SUT is improved using tests generated from the model, and the model is refined using observations made during these tests. The benefit of model-based testing is that this comparison is fully automated. And when in the beginning there is no model at all, this modification process starts from scratch with an 'empty' model, building up and 'learning' the model completely from observations that are made by applying tests to the SUT.

Example 6. Recently, we tested the new Dutch electronic passport [35]. Electronic passports contain a contactless smart-card that stores digitally-signed data including sensitive biometric data such as fingerprints. We developed a labelled transition system model of the electronic passport protocols, and used the model-based testing tool TorXakis to generate test cases and execute them on the actual passport. TorXakis is a straightforward implementation of the algorithms of the **ioco**-test theory of Sect. 3. It inherits from the model-based testing tool TorX [44], and adds symbolic test generation capabilities [19].

Although the behaviour of the passport is relatively simple, also in this case the most difficult part of the testing process was understanding the official specifications [24,17], which contain several hundreds of pages of detailed and wordy descriptions, and constructing a formal model from them. Access to electronic passports involves several protocols. In themselves, these protocols are fairly simple, yet, understanding their combination and interactions and extracting their essential behaviour, are a major challenge. Once we had a first model, the first test runs were more directed towards validating and checking the model and our understanding of the documents than towards testing the passport. Once there was sufficient confidence in the model the actual thorough testing of the passport took less than a week. The tests were run fully automatically; during a test run of a few days we were able to perform over 1,000,000 protocol steps on the passport. By refining and tweaking the model we could quickly learn how any underspecification and unclarities in the documents had been resolved in the implementation that we were testing.

Whereas for the passport we were eventually able to construct a valid model from the documentation, though with some difficulty, this was not the case for testing a wireless sensor network (WSN) node [50]. This experiment was carried out with the model-based test tools UPPAAL-TRON [30], JTORX [5], and TORXAKIS. The design and development of the WSN is mainly 'guru-driven': a few clever engineers designed and developed it, and they know how it works. This implies that making a model is driven by talking with these gurus, trying to construct a model from their explanations, and subsequently trying to get their explanations confirmed by doing model-based testing experiments on the SUT, which was one node of the WSN. In case of discrepancies between the model and the SUT we went back to the gurus trying to get more and better explanations for these discrepancies, improved the model, and re-tested the SUT. Thus, from meetings and explanations, intertwined with test experiments on the SUT, we gradually and iteratively 'learned' the behaviour of the WSN node, making modifications and additions to the model in each iteration. This process is very clarifying for the testers as well as for the guru-developers who learn more about their own system.

Test-Based Modelling. On the one hand the not infrequent practice of using model-based testing to construct and refine models, and on the other hand recent theoretical developments in automata learning and grammatical inference, have led to an interesting area of research and development on the borderline of testing, verification, and machine learning, referred to as *model learning* or *test-based modelling*. Other terms are behaviour capturing [34,32], observation-based modelling [28], or just learning.

The idea of model learning is to systematically perform experiments, or tests, on a (black-box) SUT, so that from the observations made during these tests a model can be constructed. It is a kind of black-box reverse engineering. Although the research area on grammatical inference and automata learning already exists for some time, only recently these techniques are supported by sufficiently powerful tools and have been applied successfully to learn models of software components.

The approach that is considered here is also called *active* or *adaptive* learning. It is adaptive because the tests used for obtaining observations are dynamically generated and optimized based on the information that has already been obtained during the learning process. It is active because through these tests extra observations are actively pursued as opposed to *passive* learning where a model is deduced solely from a set of existing system logs or traces without further interaction with the SUT. The latter is possible with tools like PROM [46,28].

A number of these active learning approaches have their roots in the so-called L^* algorithm of Angluin [4]. One of these developments is an adaptation of L^* for Mealy Machines, which has been implemented in the LEARNLIB tool environment [39]. In the LEARNLIB approach a teacher, who knows a Mealy Machine model M, interacts with a learner, who wishes to learn this model. Initially only knowing the sets of input and output actions, the learner asks the teacher two kind of questions: output queries asking which output occurs in response to a particular input, and equivalence queries asking whether a particular hypothesized machine H is equivalent to the machine M. If that is the case, M has been learned and the algorithm terminates; if not the learner continues with asking more output and equivalence queries. LearnLib implements an algorithm for this learning process, i.e., a recipe for the learner which output and equivalence queries to ask, to eventually know M.

If the teacher's machine M is a real SUT, then an output query can easily be answered by the teacher by performing a test on the SUT consisting of supplying the inputs to the SUT and returning the corresponding responses. An equivalence query, however, cannot be directly answered by the teacher when M is a real black-box SUT. This is where LearnLib uses model-based testing algorithms for Mealy Machines, such as the W-method and the UIO-method, to answer whether an hypothesized machine H is equivalent to the real SUT [31].

A couple of experiments have been done with LearnLib on real SUTs [23,1], among which there is also the electronic passport of Example 6, for which a model was learned from scratch and successfully compared with the one initially developed for model-based testing [2]. These experiments show the possibilities of learning in general, and of LearnLib in particular, but as identified in [3], further work is necessary, such as: (*i*) the use of abstraction techniques to learn much larger state spaces; (*ii*) using a more general model than pure Mealy Machines which require strict alternation between inputs and outputs; and (*iii*) extension to nondeterministic systems and models.

Abstraction techniques, in particular with respect to input and output actions, have been considered in [1]. An extension to (deterministic) I/O Automata [33], which do not require the strict alternation between inputs and outputs, has been studied [3]. I/O Automata are (almost) identical to the Input-Output Transition Systems $\mathcal{IOTS}$ of Sect. 3. Nondeterminism in the context of **ioco** was elaborated in [49] where the *suspension automaton* of [41] was used as a deterministic model to represent nondeterministic labelled transition systems.

In this section we will not give new algorithms for learning or work specifically on one of these extensions. In the remainder of this section model-based testing and test-based modelling will be compared and related in the context of the abstract concepts introduced in Sect. 2, with elaborations for the **ioco** theory of Sect. 3.

Learning and Model-Based Testing. Model-based testing and learning are two sides of the same coin. Both use an SUT, a model, and tests, and aim at discovering discrepancies between behaviour described in the model and behaviour exhibited by the (black-box) SUT. Model-based testing starts with the model, and a discrepancy is in the first place considered a failure of the SUT, and an incentive to modify the SUT, after which it can be retested. Learning starts with an SUT, and a discrepancy is an incentive to adapt the hypothesized model, after which the next cycle of learning can start. But as discussed above, in practical situations the difference often disappears, as on the one hand complete and valid models for model-based testing are often lacking for various reasons, and on the other hand many learning algorithms critically depend on a model-based testing step to check an hypothesized model.

Both model-based testing and learning can be described on an abstract level by the iterative process of Fig. 6. The difference between model-based testing and learning comes from a different initial model and SUT, the use of different test generation algorithms, a different choice as what to modify in case of a discrepancy, and differences in algorithmically and manually performed steps:

– In learning, the starting SUT is given and is correct by definition. A discrepancy between model and SUT leads to adapting the model.
– In learning, the initial model can be an empty or trivial model, which can be always correct or erroneous, but it can also be an initial guess, for example, a model derived using passive testing from a set of available traces, or a model obtained in a completely different way, e.g., from reverse engineering of the code.

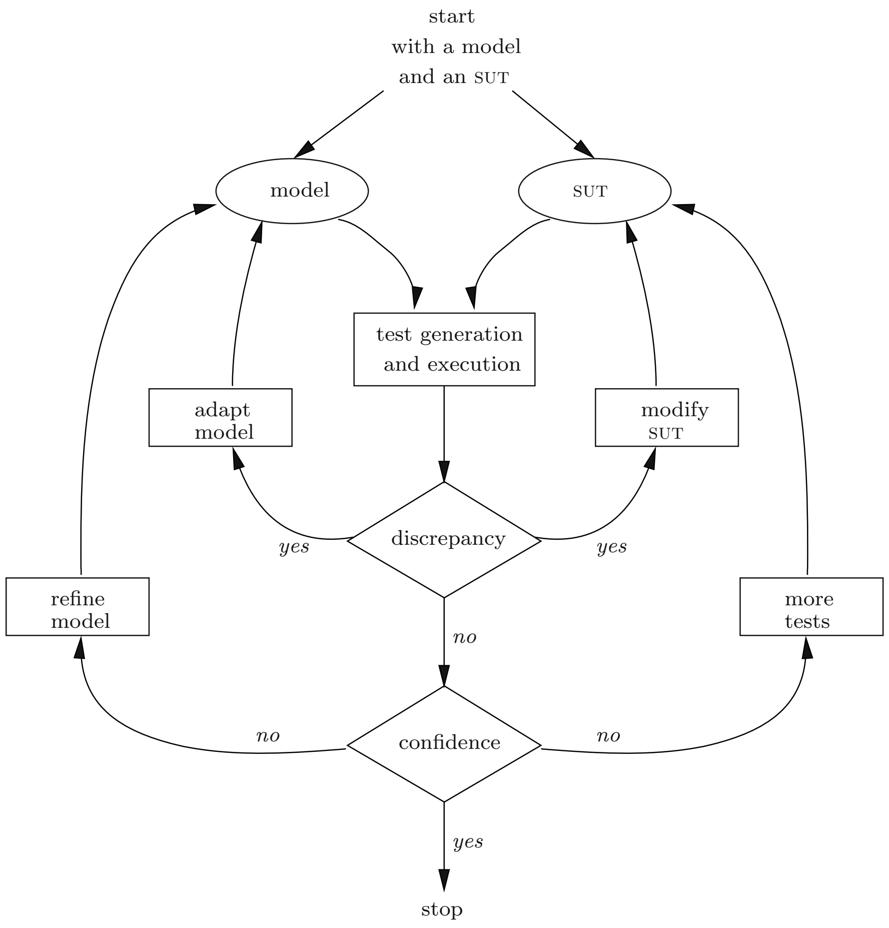


Fig. 6. The combined process of learning and model-based testing

- In learning, algorithms like Angluin-style learning take care of adapting the model completely automatically.
- A test generation algorithm for learning tries to expose as much information from the SUT as possible, i.e., it aims at rich observations with information that is useful for extending the current hypothesized model.
- If after a number of cycles in learning there is no discrepancy detected, it still can be that there is no confidence that the model is precise enough. Then the model can be refined and more tests can be executed.
- In model-based testing, the SUT is given and assumed to have been developed independently from the model.
- In (pure) model-based testing, the model is considered given and valid. Consequently, a discrepancy between model and SUT leads to modifying the SUT.
- Modifying and repairing the SUT is typically a manual step.
- A test generation algorithm for model-based testing tries to expose discrepancies between model and SUT, not necessarily providing information for extending the model.
- If after a number of cycles in model-based testing there is no discrepancy detected, it still can be that there is no confidence that the SUT is indeed correct. Then the the testing is continued with more tests.

As discussed above, current practical model-based testing processes are often a combination of learning and model-based testing. Any discrepancy has to be analysed to see whether either the model is wrong or the SUT. Such a combination completely fits in this process.

Learning in a Formal Context. We will now try to embed learning within the concepts and relations presented in Sect. 2 for model-based testing. The starting points are a formal modelling language $SPEC$, a domain of models of implementations MOD, an implementation relation **imp** $\subseteq MOD \times SPEC$, and a test execution procedure leading to observations $obs : TEST \times MOD \rightarrow \mathcal{P}(OBS)$.

LearnLib-style Learning in a Formal Context. In Angluin-style learning with LearnLib there is a model m which is in the class of deterministic, fully-specified Mealy Machines, which we denote with $\mathcal{MM}$. The goal of learning is also a deterministic, fully-specified Mealy Machine; thus we take $MOD \equiv SPEC \equiv \mathcal{MM}$. The tests that can applied to $\mathcal{MM}$ are sequences of input actions that lead to observations that are sequences of output actions; this defines $TEST$. The learning algorithm looks for a model $h \in \mathcal{MM}$ that is equivalent to m in terms of observations:

$$h \approx_{MM} m \quad \Leftrightarrow_{\text{def}} \quad \forall t \in TEST : \; obs(t,h) \; = \; obs(t,m) \tag{9}$$

This means that the learning algorithm delivers the unique model m modulo $\approx_{MM}$ (which is really unique if restricted to minimal Mealy Machines). This model can be obtained in a straightforward way using all possible tests in $TEST$ directly following (9), but fortunately LearnLib provides a more clever and efficient solution by selecting those tests that really matter based on what is already known about h.

A comparison of learning and model-based testing on $\mathcal{MM}$ shows that both are characterized by (9), the difference being the unknown variables. In testing the goal is to decide about $\approx_{MM}$ for given m and h; in learning the goal is to construct h. This leads to different algorithms, i.c. LearnLib and W- and UIO-algorithms.

Learning with Labelled Transition Systems. The next step is to consider more general models such as labelled transition systems. Such models allow arbitrary sequences of inputs and outputs instead of strict alternation, and they add nondeterminism. Arbitrary sequences of inputs and outputs were already studied in [3] in the context of I/O Automata but these systems are still deterministic.

As in Sect. 3 we consider suts behaving as possibly nondeterministic $\mathcal{IOTS}$, from which we wish to learn models that are labelled transition systems in $\mathcal{LTS}$, but not necessarily input-enabled: $MOD \equiv \mathcal{IOTS}(L_I, L_U)$ and $SPEC \equiv \mathcal{LTS}(L_I, L_U)$. We assume the signature of actions L_I, L_U to be known. Test cases in $TEST$ are as defined in Def. 8: $TEST \equiv \mathcal{TTS}(L_U, L_I)$, and observations and test execution are as introduced in Def. 9, i.e., $OBS \equiv L_\delta^*$ and test execution is given by $t\rceil|\, i \overset{\sigma}{\Longrightarrow} t'\rceil|\, i'$ where $t' = \textbf{pass}$ or $\textbf{fail}$.

When learning models in $\mathcal{LTS}$ from suts behaving as $\mathcal{IOTS}$ there are different test scenarios that can be followed depending on the relation required between the sut and the learned model. Analogous to model-based testing various relations can be chosen, and not only equivalence of models as in the LearnLib case. Just as in testing this relation is denoted by $\textbf{imp} \subseteq MOD \times SPEC$.

Though not the only choice, one obvious choice for **imp** in learning is the equivalence on $\mathcal{IOTS}$ induced by $TEST$, analogous as for LearnLib-style learning. Let $m \in \mathcal{IOTS}$ be the (model of) the sut, and let $h \in \mathcal{IOTS} \subseteq \mathcal{LTS}$ be the learned model then:

$$h \;\approx_{te}\; m \quad \Leftrightarrow_{\text{def}} \quad \forall t \in \mathcal{TTS}(L_U, L_I) :\; obs(t,h) \;=\; obs(t,m) \qquad (10)$$

This is the strongest relation that is testable on $\mathcal{IOTS}$ with $\mathcal{TTS}$. This implies that a learned model h satisfying (10) is the most precise model that can be obtained when learning m. The precision is expressed by the equivalence class $\{\, m' \in \mathcal{IOTS} \mid m' \approx_{te} m \,\}$. But most likely h is also the most expensive model, in terms of testing effort, that can be learned from m. And for nondeterministic transition systems there are no Learnlib-like algorithms yet that can construct h in a much more efficient way.

Another choice for **imp** is the **ioco** relation that is used for model-based testing. Then the goal is to learn a model $h \in \mathcal{LTS}$ such that m **ioco** h. There are many candidate models h that satisfy this goal. One such a model is m itself since m **ioco** m (reflexivity of **ioco** on $\mathcal{IOTS}$). Another candidate is the chaos system χ in Fig. 7, since for any m, m **ioco** χ. It even holds that m **ioco** χ_A for any m and any $A \subseteq L_I$; see Fig. 7. But χ and χ_A are not very precise and not distinctive so not very useful.

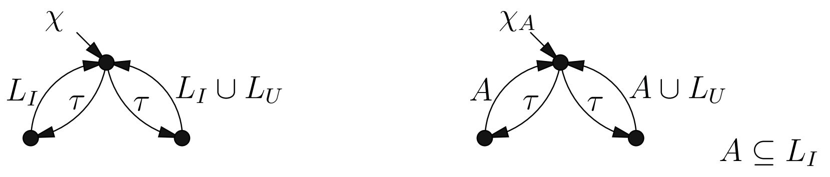


Fig. 7. Chaos. (A transition labelled with a set X is an abbreviation for all transitions with actions in that set: $q \xrightarrow{X} q' \;=_{\text{def}}\; \{q \xrightarrow{x} q' \mid x \in X\}$)

It follows that there is not a unique model that can be learned from m following the **ioco** relation but a class of models ranging from m itself to χ, and many models 'in between'. The most precise model is m itself (or more precisely its $\approx_{te}$-equivalence class), but this is also the most expensive model, in terms of test cost, to learn. The most general and therefore useless model is χ but it is also the cheapest one since it requires no testing at all. Learning can be seen as an optimization problem where we are looking for a model $h \in \mathcal{LTS}$ that optimally balances the cost of learning against obtained precision. The cost of learning a model can be expressed, for example, in the number and length of the test cases necessary to learn that model, completely analogous to Sect. 2. The precision of a learned model $h \in \mathcal{LTS}$ is related to the area of **imp**-related implementations in $\mathcal{IOTS}$. Let μ again be a monotonically increasing function on subsets of $\mathcal{IOTS}$ then the precision of a model can be expressed as $\frac{\mu(\mathcal{IOTS})-\mu(C_h)}{\mu(\mathcal{IOTS})}$ where $C_h = \{\ m' \in \mathcal{IOTS} \mid m' \ \mathbf{ioco}\ h\ \}$.

This definition of precision is analogous to the discussion on test selection for model-based testing in Sect. 2, and the measure $\mu : \mathcal{P}(MOD) \to \mathrm{I\!R}_{\geq 0}$ is the same as the one defined there. In model-based testing, see Fig. 2, the area $P_T \backslash C_s = C_{s'} \backslash C_s$ expresses the uncertainty when using an exhaustive test suite derived from s' instead of one derived from s. Completely analogously, $C_{h'} \backslash C_h$ expresses the additional uncertainty when we stop with learned model h', i.e., we stop when we know that m **ioco** h', compared to stopping at h which may be obtained if learning is continued.

Thus, when learning a specification model h from an SUT for **ioco**-based learning there are many correct learned models. These candidate models can be compared in cost of learning and in precision. Given a measure μ on $\mathcal{IOTS}$, the precision of learned specification models can be quantified and compared. Selection of the best model is a two-dimensional optimization issue, analogous to the selection of the best test suite for model-based testing in Sect. 2.

Whereas LearnLib-style learning starts with an 'empty' model, which is not correct, i.e., not equivalent, **ioco** learning can start with a correct model χ which is subsequently refined and kept correct, until a sufficiently precise model is achieved, or the additional testing for a better model becomes too costly. LearnLib-style learning, if it succeeds, gives a very precise answer based on $\mathcal{MM}$-equivalence, but it might fail due to complexity. $\mathcal{LTS}$-style learning promises better scalability because it always gives an answer, although it might be with less precision, but it is negotiable against cost of learning.

Open Issues. This section only compared model-based testing and learning on an abstract level. No concrete algorithms for $\mathcal{LTS}$ or **ioco**-learning have been presented. Such an algorithm is, for example, presented in [49], but further elaborations are necessary.

Another one of our abstractions that obviously needs concretization is the use of the measure μ in order to quantify the over-approximation and uncertainty in testing and learning. As explained, this measure is equally applicable to expressing the quality of learned models, as well as to expressing the quality of selected test suites. Such a measure can, for example, be defined as a measure-theoretic integral over the space $\mathcal{IOTS}$ [9], but this is far from trivial. As shown in Sect. 2 such a measure can be replaced by a distance function on specification models. First attempts in this direction are [13,18].

When learning and model-based testing are combined care should be taken how to use the learned models. Doing model-based testing on a system with a fully learned model of the same system does not make sense. Testing requires independence: the SUT and the test cases, or the model from which the test cases were generated, should have been developed independently. Yet, such a learned model can be used for regression testing, i.e., testing whether a modified component still complies with the old specification. Another application domain is testing of refactored or re-implemented legacy components, which is a growing area of interest. In addition learned models can be used for other design and analysis activities, such as increasing understanding about systems, communication among stakeholders, model-based analysis of properties, model checking, and simulation.

It looks natural to extend learning of nondeterministic models with probabilities indicating the occurrence frequency of nondeterministic transitions. This opens the way towards combining with the rich area of probabilistic and stochastic state-based models.

A last point that is mentioned is uncertainty in LearnLib-style learning. This style of learning depends for its equivalence query for a real SUT on a model-based test. Model-based testing of Mealy Machines does not provide a complete decision procedure. In particular, completeness depends on the assumption that the number of states of the SUT is smaller than or equal to the number of states of the hypothesized model h. This assumption may be violated, so that testing is not complete, and consequently, the learned model is not equivalent to the SUT. Additional investigations are necessary to deal with, and quantify such incompleteness and uncertainty.

Acknowledgements. I wish to thank Fides Aarts, Brian Nielsen, Lars Frantzen, and Frits Vaandrager for discussions and for comments on earlier versions of some of the sections.

References

1. Aarts, F.: Inference and Abstraction of Communication Protocols. Master's thesis, Institute for Computing and Information Sciences, Radboud University, and Uppsala University, Nijmegen, The Netherlands, and Uppsala, Sweden (2009)

2. Aarts, F., Schmaltz, J., Vaandrager, F.: Inference and abstraction of the biometric passport. In: Margaria, T., Steffen, B. (eds.) ISoLA 2010. LNCS, vol. 6415, pp. 673–686. Springer, Heidelberg (2010)
3. Aarts, F., Vaandrager, F.: Learning I/O Automata. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010. LNCS, vol. 6269, pp. 71–85. Springer, Heidelberg (2010)
4. Angluin, D.: Learning Regular Sets from Queries and Counterexamples. Information and Computation 75(2), 87–106 (1987)
5. Belinfante, A.: JTorX: A Tool for On-Line Model-Driven Test Derivation and Execution. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 266–270. Springer, Heidelberg (2010)
6. Berg, T., Jonsson, B., Raffelt, H.: Regular Inference for State Machines with Parameters. In: Baresi, L., Heckel, R. (eds.) FASE 2006. LNCS, vol. 3922, pp. 107–121. Springer, Heidelberg (2006)
7. Bernot, G., Gaudel, M.G., Marre, B.: Software testing based on formal specifications: a theory and a tool. Software Engineering Journal, 387–405 (November 1991)
8. Bollig, B., Katoen, J.P., Kern, C., Leucker, M.: Replaying Play in and Play out: Synthesis of Design Models from Scenarios by Learning. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 435–450. Springer, Heidelberg (2007)
9. Brinksma, E.: On the coverage of partial validations. In: Nivat, M., Rattray, C., Rus, T., Scollo, G. (eds.) AMAST 1993. BCS-FACS Workshops in Computing Series, pp. 247–254. Springer, Heidelberg (1993)
10. Brinksma, E., Tretmans, J., Verhaard, L.: A framework for test selection. In: Jonsson, B., Parrow, J., Pehrson, B. (eds.) Protocol Specification, Testing, and Verification XI, pp. 233–248. North-Holland, Amsterdam (1991)
11. Broy, M., Jonsson, B., Katoen, J.P., Leucker, M., Pretschner, A. (eds.): Model-Based Testing of Reactive Systems. LNCS, vol. 3472. Springer, Heidelberg (2005)
12. Campbell, C., Grieskamp, W., Nachmanson, L., Schulte, W., Tillmann, N., Veanes, M.: Testing Concurrent Object-Oriented Systems with SPEC EXPLORER – Extended Abstract. In: Fitzgerald, J., Hayes, I., Tarlecki, A. (eds.) FM 2005. LNCS, vol. 3582, pp. 542–547. Springer, Heidelberg (2005)
13. Curgus, J., Vuong, S.: Sensitivity analysis of the metric based test selection. In: Kim, M., Kang, S., Hong, K. (eds.) Int. Workshop on Testing of Communicating Systems, vol. 10, pp. 200–219. Chapman & Hall, Boca Raton (1997)
14. De Nicola, R.: Extensional Equivalences for Transition Systems. Acta Informatica 24, 211–237 (1987)
15. De Nicola, R., Hennessy, M.: Testing Equivalences for Processes. Theoretical Computer Science 34, 83–133 (1984)
16. Dijkstra, E.: Notes On Structured Programming, End of section 3: On The Reliability of Mechanisms (1969)
17. Advanced Security Mechanisms for Machine Readable Travel Documents – Extended Access Control (EAC) – Version 1.11. Tech. Rep. TR-03110, German Federal Office for Information Security (BSI), Bonn, Germany (2008)
18. Feijs, L., Goga, N., Mauw, S., Tretmans, J.: Test Selection, Trace Distance and Heuristics. In: Schieferdecker, I., König, H., Wolisz, A. (eds.) Testing of Communicating Systems XIV, pp. 267–282. Kluwer Academic Publishers, Dordrecht (2002)
19. Frantzen, L., Tretmans, J., Willemse, T.: Test Generation Based on Symbolic Specifications. In: Grabowski, J., Nielsen, B. (eds.) FATES 2004. LNCS, vol. 3395, pp. 1–15. Springer, Heidelberg (2005)

20. Gaudel, M.C.: Testing can be formal, too. In: Mosses, P., Nielsen, M., Schwartzbach, M. (eds.) TAPSOFT 1995. LNCS, vol. 915, pp. 82–96. Springer, Heidelberg (1995)
21. Grieskamp, W.: Microsoft's protocol documentation program: A success story for model-based testing. In: Bottaci, L., Fraser, G. (eds.) TAIC PART 2010. LNCS, vol. 6303, pp. 7–7. Springer, Heidelberg (2010)
22. Groz, R., Charles, O., Renévot, J.: Relating Conformance Test Coverage to Formal Specifications. In: Gotzhein, R. (ed.) FORTE 1996. Chapman & Hall, Boca Raton (1996)
23. Hungar, H., Margaria, T., Steffen, B.: Domain-Specific Optimization in Automata Learning. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 315–327. Springer, Heidelberg (2003)
24. Doc 9303 – Machine Readable Travel Documents – Part 1–2. Tech. rep., ICAO, 6 edn (2006),
25. Jacky, J., Veanes, M., Campbell, C., Schulte, W.: Model-Based Software Testing and Analysis with C#. Cambridge University Press, Cambridge (2008)
26. Jard, C., Jéron, T.: TGV: Theory, Principles and Algorithms: A Tool for the Automatic Synthesis of Conformance Test Cases for Non-Deterministic Reactive Systems. Software Tools for Technology Transfer 7(4), 297–315 (2005)
27. Jeannet, B., Jéron, T., Rusu, V., Zinovieva, E.: Symbolic Test Selection based on Approximate Analysis. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 349–364. Springer, Heidelberg (2005)
28. Kanstrén, T., Piel, E., Gonzalez-Sanchez, A., Gross, H.G.: Observation-Based Modeling for Testing and Verifying Highly Dependable Systems – A Practitioner's Approach. In: Wagner, A. (ed.) Workshop on Design of Dependable Critical Systems at Safecomp 2009, Hamburg, Germany (September 2009)
29. Koopman, P., Alimarine, A., Tretmans, J., Plasmeijer, R.: Gast: Generic Automated Software Testing. In: Peña, R., Arts, T. (eds.) IFL 2002. LNCS, vol. 2670, pp. 84–100. Springer, Heidelberg (2003)
30. Larsen, K., Mikucionis, M., Nielsen, B.: Online Testing of Real-Time Systems using Uppaal. In: Grabowski, J., Nielsen, B. (eds.) FATES 2004. LNCS, vol. 3395, pp. 79–94. Springer, Heidelberg (2005)
31. Lee, D., Yannakakis, M.: Principles and Methods for Testing Finite State Machines – A Survey. The Proceedings of the IEEE 84(8), 1090–1123 (1996)
32. Lorenzoli, D., Mariani, L., Pezzè, M.: Automatic generation of software behavioral models. In: ICSE 2008: 30th Int. Conf. on Software Engineering, pp. 501–510. ACM, New York (2008)
33. Lynch, N.: Distributed Algorithms. Morgan Kaufmann Publishers Inc., San Francisco (1996)
34. Mariani, L., Pezzè, M.: Behaviour Capture and Test: Automated Analysis of Component Integration. In: 10th IEEE Int. Conf. on Engineering of Complex Computer Systems – ICECCS 2005, pp. 292–301. IEEE Computer Society, Los Alamitos (2005)
35. Mostowski, W., Poll, E., Schmaltz, J., Tretmans, J., Wichers Schreur, R.: Model-Based Testing of Electronic Passports. In: Alpuente, M., Cook, B., Joubert, C. (eds.) FMICS 2009. LNCS, vol. 5825, pp. 207–209. Springer, Heidelberg (2009)
36. Oostdijk, M., Rusu, V., Tretmans, J., de Vries, R., Willemse, T.C.: Integrating Verification, Testing, and Learning for Cryptographic Protocols. In: Davies, J., Gibbons, J. (eds.) IFM 2007. LNCS, vol. 4591, pp. 538–557. Springer, Heidelberg (2007)

37. Peled, D., Vardi, M., Yannakakis, M.: Black Box Checking. Journal of Automata, Languages, and Combinatorics 7(2), 225–246 (2002)
38. Petrenko, A.: Fault Model-Driven Test Derivation from Finite State Models: Annotated Bibliography. In: Cassez, F., Jard, C., Rozoy, B., Dermot, M. (eds.) MOVEP 2000. LNCS, vol. 2067, pp. 196–205. Springer, Heidelberg (2001)
39. Raffelt, H., Merten, M., Steffen, B., Margaria, T.: Dynamic testing via automata learning. Software Tools for Technology Transfer 11(4), 307–324 (2009)
40. Raffelt, H., Steffen, B., Berg, T., Margaria, T.: LearnLib: A framework for extrapolating behavioral models. Software Tools for Technology Transfer 11(5), 393–407 (2009)
41. Tretmans, J.: Test Generation with Inputs, Outputs and Repetitive Quiescence. Software—Concepts and Tools 17(3), 103–120 (1996)
42. Tretmans, J. (ed.): Tangram: Model-Based Integration and Testing of Complex High-Tech Systems. Embedded Systems Institute, Eindhoven (2007), `http://www.esi.nl/publications/tangramBook.pdf`
43. Tretmans, J.: Model Based Testing with Labelled Transition Systems. In: Hierons, R., Bowen, J., Harman, M. (eds.) FORTEST. LNCS, vol. 4949, pp. 1–38. Springer, Heidelberg (2008)
44. Tretmans, J., Brinksma, E.: TorX : Automated Model Based Testing. In: Hartman, A., Dussa-Zieger, K. (eds.) First European Conference on Model-Driven Software Engineering, Imbuss, Möhrendorf, Germany, December 11-12 (2003)
45. Utting, M., Legeard, B.: Practical Model-Based Testing: A Tools Approach. Morgan-Kaufmann, San Francisco (2007)
46. Verbeek, E., Buijs, J., van Dongen, B., van de Aalst, W.: Prom 6: The Process Mining Toolkit. In: 8th Int. Conf. on Business Process Management – BPM 2010 (2010)
47. de Vries, R., Belinfante, A., Feenstra, J.: Automated Testing in Practice: The Highway Tolling System. In: Schieferdecker, I., König, H., Wolisz, A. (eds.) Testing of Communicating Systems XIV, pp. 219–234. Kluwer Academic Publishers, Dordrecht (2002)
48. de Vries, R., Tretmans, J.: Towards Formal Test Purposes. In: Brinksma, E., Tretmans, J. (eds.) Formal Approaches to Testing of Software – FATES 2001. BRICS Notes Series, vol. NS-01-4, pp. 61–76. BRICS, University of Aarhus, Denmark (2001)
49. Willemse, T.: Heuristics for ioco-Based Test-Based Modelling. In: Brim, L., Haverkort, B., Leucker, M., van de Pol, J. (eds.) FMICS 2006 and PDMC 2006. LNCS, vol. 4346, pp. 132–147. Springer, Heidelberg (2007)
50. Zhu, F.: Testing Timed Systems in Simulated Time with Uppaal-Tron: An Industrial Case Study. Master's thesis, Institute for Computing and Information Sciences, Radboud University, Nijmegen, The Netherlands (2010)

Learning of Automata Models Extended with Data[⋆]

Bengt Jonsson

Department of Computer Systems, Uppsala University, Sweden
bengt@it.uu.se

Abstract. One of the challenges in the CONNECT project is to develop techniques for learning models of networked components from exploratory interaction with the component, based on analyzing messages exchanged between the component and its environment. Many approaches to this problem employ regular inference (aka. automata learning) techniques which generate modest-size finite-state models. Most communication with real-life systems involves data values being relevant to the communication context and thus influencing the observable behavior of the communication endpoints. When applying methods from the realm of automata learning, it is desirable to handle such data-occurrences. It is therefore important to extend inference techniques to handle message alphabets and state-spaces with structures containing data parameters, often with large domains. After very briefly mentioning several approaches to the problem, we give a longer account of an approach proposed by Aarts et al, which adapts ideas from of predicate abstraction, successfully used in formal verification. We illustrate the techniques by application to a simple running example, which models a simple booking service.

1 Introduction

Interoperability remains a fundamental challenge when connecting heterogeneous systems which encounter and spontaneously communicate with one another in pervasive computing environments. The CONNECT Integrated Project [22] aims at overcoming the interoperability barrier by synthesizing on the fly the CONNECTors via which networked systems communicate. CONNECTors are implemented through a comprehensive dynamic process based on (i) extracting knowledge from, (ii) learning about and (iii) reasoning about, the interaction behavior of networked systems, together with (iv) synthesizing new interaction behaviors out of the ones exhibited by the systems to be made interoperable.

One of the challenges in the CONNECT project is to develop techniques for learning models of exploratory interaction with the component, based on analyzing messages exchanged between the component and its environment. Generation of models by exploratory interaction can be useful also in other contexts.

⋆ Supported in part by EC Proj. 231167 (CONNECT).

A large source of application might be found in model-based verification and validation, including model checking and model-based testing [7]. Such techniques have witnessed drastic advances in the last decades, and are being applied to verification and valiation of communication protocols, hardware systems, embedded controllers, etc., also in industrial settings (e.g., [20]). They require models that specify the intended behavior of system components, which ideally should be developed during specification and design. However, the construction of models typically requires significant manual effort, implying that in practice often models are not available, or become outdated as the system evolves. Automated support for constructing models of the behavior of implemented components would therefore be useful also, e.g., for regression testing, for replacing manual testing by model based testing, for producing models of standardized protocols, for analyzing whether an existing system is vulnerable to attacks, etc.

The construction of models from observations of component behavior can be performed using automata learning (aka. regular inference) techniques [4, 11, 13, 23, 30, 33]. This class of techniques is now receiving increasing attention in the testing and verification community, e.g., for regression testing of telecommunication systems [18, 21], for integration testing [17, 24], security protocol testing [32], and for combining conformance testing and model checking [29, 16]. One of the most used algorithms for regular inference, L^*, is thoroughly explained in the Chapter *Introduction to Active Automata Learning from a Practical Perspective* by Steffen, Howar, and Merten. This algorithm poses a sequence of *membership queries*, each of which observes the component's output in response to a certain input word, and produces a minimal deterministic finite-state machine which conforms to the observations. If the sequence of membership queries is sufficiently large, the produced machine will be a model of the observed component.

Since regular inference techniques are designed for finite-state models, most previous applications to model generation have been limited to generating control flow skeletons, suppressing data which appear, e.g., as parameters of messages. However, data parameters have a significant impact on control flow and behavior in typical networked components and protocol entities. they can be sequence numbers, configuration parameters, agent and session identifiers, etc.; a model of a networked service is considerabely less informative if information about exchanged data is suppressed. It is therefore important to extend inference techniques to handle message alphabets and state-spaces with structures containing data parameters with large domains.

In this chapter, we will consider the problem of extending learning to automata with data, by presenting a particular approach, introduced in the work by Aarts, Jonsson, Uijen, and Vaandrager [1, 2]. We first define a model for symbolic representation of protocols. Thereafter, we present a technique for using the L^* algorithm, designed for inference of finite-state Mealy machines, to infer also symbolically defined protocol models. The technique is inspired by predicate abstraction [26, 9], which has been successful for extending finite-state model checking to large and infinite state spaces. In contrast to that work, however, we are now in a black-box setting, where an abstraction cannot be defined based on the source code or model of a component, since it is not accessible. Instead, we must construct an externally supplied abstraction, which translates between a large message alphabet of the component to be modeled and a small finite alphabet of the regular inference algorithm. Via regular inference, a finite-state model of the abstracted interface is inferred. The abstraction can then be reversed to generate a faithful model of the component.

The presented approach was used to learn models of reduced versions of the SIP and TCP protocols, in [1], and also to learn a model of the new generation of biometric passports in [2]. A We will also describe how to construct a suitable abstraction, utilizing pre-existing knowledge about which operators are sufficient to express guards and operations on data in a faithful model of the component.

On Related Work. Regular inference techniques have been used for several tasks in verification and test generation, e.g., to create models of environment constraints with respect to which a component should be verified [10], for regression testing to create a specification and test suite [18, 21], to perform model checking without access to source code or formal models [16, 29], for program analysis [3], and for formal specification and verification [10].

In several approaches, the challenge of including data parameters of message have been addressed. In the work of Shu and Lee [32], parameters are essentially suppressed in order to obtain a finite subset of input symbols when learning the behavior of security protocol implementations. This subset can be extended in response to new information obtained in counterexamples. Groz, Li, and Shahbaz [24, 31, 17] extend regular inference to Mealy machines with data values, for use in integration testing. In their work, they select a finite set of representative data values to be supplied together with the input to ta component.

An influential approach to learning properties of data in programs is represented by the Daikon system [12]. Its basic technique is to observe executions of a component, and extract invariants over program variables that are observed to hold. The invariants can be chosen from a predefined collection. The Daikon system does not immediately consider to extract control structures of components. There are several approaches that combine regular inference for learning control structures, and the Daikon tool (or similar) for inferring constraints on data parameters. One of the questions to be solved in such a combination is how to correlate the two types of models.

Lorenzoli, Mariani, and Pezzé infer models of software components that consider both sequence of method invocations and their associated data parameters [27, 28]. They use a passive learning approach where a finite control structure that captures possible sequences of method invocations is infer by an extension of the k-tails algorithm (a passive learning algorithm), and using Daikon [8] to infer guards and relations on method parameters. This allows to infer constraints on data parameters that are exchanged after specific sequences of method invocations, but not to analyze the influence of data parameter on subsequent control behavior. The same basic combination is also employed by Lo and Maoz [25], which infer a more refined view on constraints over data parameters, in that different constraints are generated for different scenarios, if a need for this is detected.

In previous work, we have considered extensions of regular inference to handle data parameters. In [5], we show how guards on boolean parameters can be refined lazily. This technique for maintaining guards have inspired the more general notion of abstractions on input symbols presented in this chapter. We have also proposed techniques to handle infinite-state systems, in which parameters of messages and state variables are from an unbounded domain, e.g., for identifiers [6], and timers [15, 14]. These extensions are specialized towards a particular data domain, and their worst-case complexities do not immediately suggest an efficient implementation.

Organization. In the next section, we first introduce a simple running example that will serve to illustrate the techniques presented in this tutorial. Thereafter, we introduce Mealy machines, and our symbolic extension of Mealy machines, that include data. The technique of using abstraction to adapt finite-state learning algorithms to symbolically defined Mealy machines is presented in Section 5. This techniques requires an abstraction which in general must be constructed manually. A technique for systematic construction of such abstractions is presented in 7. We illustrate the application of this technique to the running example in Section 6.

2 A Running Example

Let us introduce a small example to illustrate the techniques that will be introduced in later sections. Imagine a service for booking seats in a concert or similar event. A user of this service has to provide his credentials and can then browse through a list of seats. From the list of seats, a single seat can be booked, which will be confirmed in a corresponding receipt.

Imagine further that an *a priori* interface description of the service is provided, specifying a specific set of messages that are understood by the service, containing

– openSession with two parameters, a user name and a password, supplies credentials, and if they are accepted the service provides a session identifier in response,
– getSeats with a session identifier as parameter, asks for a list of avaible seats that can be booked,
– getSeat with a session identifier and a seat as parameters, asks to book a specific seat, and if accepted, the service will confirm by a positive reply.

The exchange of interface primitives during a typical session can be informally depicted in sequence chart in Figure 1.

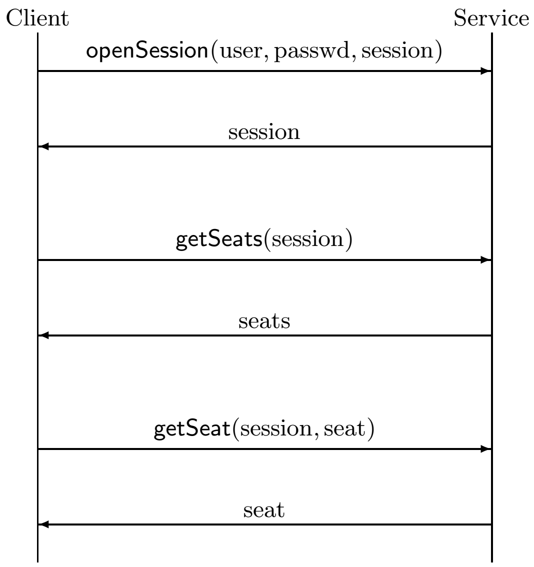


Fig. 1. Message Flow in the Running Example

In the following sections, we will consider how active automata learning, which is able to generate finite-state Mealy machines from queries, can be used to generate a model of the booking service.

3 Mealy Machines

Throughout the presentation, we will use *Mealy machines* to model the behavior of communication protocol entities, networked services, etc.

Definition 1. A *Mealy machine* is a tuple $\mathcal{M} = \langle \Sigma_I, \Sigma_O, Q, q_0, \delta, \lambda \rangle$ where

- Σ_I is a nonempty set of *input symbols*,
- Σ_O is a nonempty set of *output symbols*,
- Q is a nonempty set of *states*,
- $q_0 \in Q$ is the *initial state*,
- $\delta : Q \times \Sigma_I \to Q$ is the *transition function*, and
- $\lambda : Q \times \Sigma_I \to \Sigma_O$ is the *output function*. □

The sets of states and symbols can be finite or infinite: if they are both finite we say that the Mealy machine is *finite-state*. Elements of Σ_I^* are called *input words*, and elements of Σ_O^* are called *output words*.

Intuitively, a Mealy machine behaves as follows. At any point in time, the machine is in some state $q \in Q$. When supplied with an input symbol $a \in \Sigma_I$, it responds by producing an output symbol $\lambda(q, a)$ and transforms itself to a new state $\delta(q,a)$. We use the notation $q \xrightarrow{a/b} q'$ to denote that $\delta(q,a) = q'$ and $\lambda(q,a) = b$; in this case $q \xrightarrow{a/b} q'$ is called a *transition* of $\mathcal{M}$.

We can depict Mealy machines as directed edge-labeled graphs, where Q is the set of vertices. The outgoing edges from a state $q \in S$ lead to $\delta(q,a)$ for all $a \in \Sigma_I$, and they are labeled "a/b", where a is the input symbol and b is the output symbol $\lambda(q,a)$. As an example, Figure 2 shows a Mealy machine that receives a sequence of symbols of form a or b. Whenever an a-symbol is received, it outputs the number of received a-symbols modulo 2, and whenever a b-symbol is received, it outputs the number of received a-symbols modulo 4. The initial state is q_0.

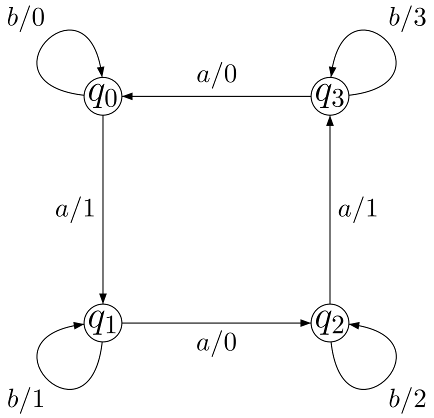


Fig. 2. A Mealy machine $\langle \Sigma_I, \Sigma_O, Q, q_0, \delta, \lambda\rangle$ with states $Q = \{q_1, q_2, q_3, q_4\}$, input alphabet $\Sigma_I = \{a, b\}$, and output alphabet $\Sigma_O = \{0, 1, 2, 3\}$. For instance, applying a starting in q_0 produces output $\lambda(q_0, a) = 1$ and moves to next state $\delta(q_0, a) = q_1$.

Applying a word $a_1 a_2 \cdots a_k \in \Sigma_I^*$ of input symbols starting in a state q_0 results in the sequence of states $q_0, q_1, \ldots, q_k$ with $q_j = \delta(q_{j-1}, a_j)$ for $j = 1, \ldots, k$. We extend the transition function to $\delta(q_0, a_1 a_2 \cdots a_k) \overset{\text{def}}{=} q_k$ and the output function to $\lambda(q_0, a_1 a_2 \cdots a_k) \overset{\text{def}}{=} \lambda(q_0, a_1)\lambda(q_1, a_2) \cdots \lambda(q_{k-1}, a_k)$, i.e., the concatenation of all outputs. We define $\lambda_{\mathcal{M}}(u) = \lambda(q_0, u)$ for $u \in \Sigma_I^*$. Two Mealy machines $\mathcal{M}$ and $\mathcal{M}'$ with the same set of input symbols are *equivalent* if $\lambda_{\mathcal{M}}(u) = \lambda_{\mathcal{M}'}(u)$ for all input words u.

Mealy machines are *completely specified*, meaning that at every state there is a next state for every input (δ and λ are total). They are also *deterministic*, because only one next state is possible.

4 Symbolic Mealy Machines

Finite-state Mealy machines, introduced in the previous section, cannot represent all aspects of the behavior of protocols and networked components, where the interplay between control and data is significant. Typical such models have an infinite number of states and infinite communication alphabets that span domains of data values that are very large or infinite. Typical examples of such data domains are integers to represent sequence numbers, session identifiers, etc., strings to represent exchanged data, etc.

In this section, we introduce *Symbolic Mealy Machines.* They can be seen as a symbolic representation of large or infinite-state Mealy machines, in that input and output symbols have parameters which are data values, e.g., to represent messages in a typical communication protocol. Often, data parameters are from rather large (in practice "infinite") domains, and on which rather simple operations and tests are applied, e.g., equality tests between elements of the same domain, or a check whether an element is a member of some set. It seems reasonable to be able to extend automata learning to such models, in analogy with the way automated verification techniques have been extended from finite-state models to extensions that cover, e.g., clocks as in timed automata.

Data Values. We first consider the data values that occur as parameters of input and output symbols and stored in state variables that are part of the representation of the internal state of a Mealy machine. We will use d, d_1, d_2, etc. to range over data values. To describe the data values that are relevant for a symbolic Mealy machine, we assume a finite set of *domains*, each of which is a (finite or infinite) set of data values. We also assume a finite set of *functions* and a finite set of *predicates*. Each function f has an *arity*, denoted $\mathcal{D}_1 \times \cdots \times \mathcal{D}_n \mapsto \mathcal{D}$, where $\mathcal{D}_1, \ldots, \mathcal{D}_n$ and $\mathcal{D}$ are domains, meaning that the arguments to f must be an n-tuple of data values $d_1, \ldots, d_n$, where $d_i \in \mathcal{D}_i$ for $i = 1, \ldots, n$, and then $f(d_1, \ldots, d_n)$ is an element in $\mathcal{D}$. We write $f : \mathcal{D}_1 \times \cdots \times \mathcal{D}_n \mapsto \mathcal{D}$ to denote that f has arity $\mathcal{D}_1 \times \cdots \times \mathcal{D}_n \mapsto \mathcal{D}$. A predicate r has an arity, denoted $\mathcal{D}_1 \times \cdots \times \mathcal{D}_n$, meaning that it can be thought of as a function from $\mathcal{D}_1 \times \cdots \times \mathcal{D}_n$ to boolean values.

Input and Output Symbols. Input and output symbols will be represented using finite sets I and O of (input and output) *ations.* Each ation α has a certain *arity*, which is a tuple of *domains* $\mathcal{D}_{\alpha,1}, \ldots, \mathcal{D}_{\alpha,n}$ (where n depends on α). Let Σ_I be the set of *input symbols* of form $\alpha(d_1, \ldots, d_n)$, where $d_i \in \mathcal{D}_{\alpha,i}$ is in the appropriate domain for each i with $1 \leq i \leq n$. The set of *output symbols* Σ_O is defined analogously.

Example. For the service introduce in Section 2, we use the following domains to represent the data values that occur in input and output symbols.

- STRING contains data values for user names and passwords.
- SESSION contains identifiers of sessions: it could be, e.g., the set of natural numbers.
- SEAT contains the possible seats (e.g., represented by seat numbers) that are available in the event, and
- SEATS contains sets of seats in SEAT.

We use the following predicates.

- $\in$: $\mathsf{SEAT} \times \mathsf{SEATS}$ is the test for membership, and
- has_passwd : $\mathsf{STRING} \times \mathsf{STRING}$ tests for valid combinations of usernames and passwords.

In addition, we include the equality predicate $=$ on the domains $\mathsf{SESSION}$ and SEAT.

The set of input ations with corresponding arities is described in the following table.q

Input ation	arity
openSession :	$\mathsf{STRING}, \mathsf{STRING}, \mathsf{SESSION}$
getSeats :	$\mathsf{SESSION}$
getSeat :	$\mathsf{SESSION}, \mathsf{SEAT}$

To model the response from the service, we could use one output ation for each kind of response, e.g., an ation returnSession with arity $\mathsf{SESSION}$ for replies to input symbols of form openSession(u, p, s). To save space, we will simply just let the reply be modeled by a data element from the appropriate domain. □

Symbolic Mealy Machines. We can now define symbolic Mealy machines. We assume a set of domains, functions, and predicates, as described in the previous paragraphs, which will be used to form expressions denoting data values, and boolean expressions to denote tests on data values. We assume that expressions always follow the restrictions of the relevant arities.

We assume a set of *formal parameters*, ranged over by $p_1, p_2, \ldots$, to be used as placeholders for parameters of symbols in symbolic transitions. We also assume a set of *state variables*, each with a domain of possible values, and a unique initial value.

Definition 2. A *Symbolic Mealy machine* (SMM for short) is a tuple $\mathcal{SM} = \langle I, O, L, l_0, X, \longrightarrow \rangle$, where

- I and O are disjoint finite sets of actions (*input ations* and *output ations*),
- L is a finite set of *locations*,
- $l_0 \in L$ is the *initial location*,
- X is a finite set of state variables; each state variable x has a domain $\mathcal{D}_x$ of possible values, and a unique initial value, and
- $\longrightarrow$ is a finite set of *symbolic transitions*, each of form

$$\textcircled{l} \xrightarrow{\alpha(p_1, \ldots, p_n) \ \mathbf{when} \ g \ / \ x_1, \ldots, x_k := e_1, \ldots, e_k \ ; \ \beta({e^{out}}_1, \ldots, {e^{out}}_m)} \textcircled{l'}$$

  in which
  - l and l' are locations,
  - $\alpha \in I$ and $\beta \in O$ are input and output actions,
  - $p_1, \ldots, p_n$ are distinct formal parameters,
  - $x_1, \ldots, x_k$ are distinct state variables in X,
  - g (the *guard*) is a boolean expression over the formal parameters $p_1, \ldots, p_n$ and the state variables in X, and

- $e_1, \ldots, e_k$ and ${e^{out}}_1, \ldots, {e^{out}}_m$ are tuples of expressions over $p_1, \ldots, p_n$ and X. We assume that the arities of α and β and the domains of $x_1, \ldots, x_k$ are respected. □

Intuitively, a symbolic transition of the above form denotes steps of the Mealy machine in which some input symbol of form $\alpha(d_1, \ldots, d_n)$ is received, whereby the formal parameters $p_1, \ldots, p_n$ are bound to the received data values $d_1, \ldots, d_n$; in case the guard g is evaluated to true, the state variables among $x_1, \ldots, x_k$ are assigned new values by the assignment $x_1, \ldots, x_k := e_1, \ldots, e_k$, and an output symbol, obtained by evaluating $\beta({e^{out}}_1, \ldots, {e^{out}}_m)$, is generated. In case the guard g is evaluted to false, then the symbolic transition does not denote any step.

Semantics of SMM. We can give a precise meaning to an SMM by letting it denote a Mealy machine with possibly infinite sets of input and output symbols and states. Such a Mealy machine can be defined as follows. Assume an SMM $\mathcal{SM}$ defined as the tuple $\langle I, O, L, l_0, X, \longrightarrow \rangle$. A *valuation* is an assignment σ which maps each location variable x in X to a data value in $\mathcal{D}_x$. Valuations are extended to expressions in the natural way: for instance, if $\sigma(x_3) = 8$, then $\sigma(2 * x_3 + 4) = 20$. We let σ_0 denote the valuation which maps each variable to its initial value. In the following, we will use $\overline{p}$ for $p_1, \ldots, p_n$ and $\overline{d}$ for $d_1, \ldots, d_n$.

Definition 3. We define a SMM $\mathcal{SM} = \langle I, O, L, l_0, X, \longrightarrow \rangle$ as denoting a (typically infinite-state) Mealy machine $\mathcal{M}_{\mathcal{SM}} = \langle \Sigma_I, \Sigma_O, Q, q_0, \delta, \lambda \rangle$, where

– Σ_I is obtained from I as described when introducing actions, and similarly for Σ_O,
– Q is the set of pairs $\langle l, \sigma \rangle$ consisting of a location $l \in L$ and a valuation σ,
– q_0 is the pair $\langle l_0, \sigma_0 \rangle$, and
– δ and λ are such that for any symbolic transition in $\longrightarrow$ of form

$$\begin{tikzcd} \textcircled{l} \arrow[r, "\alpha(\overline{p})\ \textbf{when}\ g\ /\ x_1, \ldots, x_k := e_1, \ldots, e_k\ ;\ \beta({e^{out}}_1, \ldots, {e^{out}}_m)"] & \textcircled{l'} \end{tikzcd}$$

for any valuation σ and data values $\overline{d}$ such that $\sigma(g[\overline{d}/\overline{p}])$ is true (i.e., under the valuation σ, the guard g is evaluated to true when the formal parameters $\overline{p}$ are replaced by the received data values $\overline{d}$), it holds that

- $\delta(\langle l, \sigma \rangle, \alpha(\overline{d})) = \langle l', \sigma' \rangle$, where σ' is the valuation such that
  - $\sigma'(x_i) = \sigma(e_i[\overline{d}/\overline{p}])$ for $1 \leq i \leq k$, and
  - $\sigma'(x) = \sigma(x)$ if x is not among $x_1, \ldots, x_k$,
- $\lambda(\langle l, \sigma \rangle, \alpha(\overline{d})) = \beta(\sigma'(e_1^{out}[\overline{d}/\overline{p}]), \ldots, \sigma'(e_m^{out}[\overline{d}/\overline{p}]))$. □

Here the four last lines say that the state is updated to a new pair $\langle l', \sigma' \rangle$, where l' is the target location of the symbolic transition and σ' is obtained by performing the multiple assignment $x_1, \ldots, x_k := e_1, \ldots, e_k$ simultaneously to all variables that are among $x_1, \ldots, x_k$, and that an output symbol, obtained by evaluating the expression $\beta({e^{out}}_1, \ldots, {e^{out}}_m)$ in σ, is generated.

Having defined the meaning of an SMM through translation to an ordinary Mealy machine, we can inherit some definitions. We use $\lambda_{\mathcal{SM}}$ to denote $\lambda_{\mathcal{M}_{\mathcal{SM}}}$, and say that $\mathcal{SM}$ and $\mathcal{SM}'$ are equivalent if $\lambda_{\mathcal{SM}}(u) = \lambda_{\mathcal{SM}'}(u)$ for all input words u. We can similarly say that an SMM is equivalent to a Mealy machine.

Symbolic Mealy machines are required to be deterministic, just like ordinary Mealy machines. We say that $\mathcal{SM}$ is deterministic if $\mathcal{M}_{\mathcal{SM}}$ is deterministic: a sufficient condition under which $\mathcal{M}_{\mathcal{SM}}$, is deterministic is that for each input action $\alpha \in I$, each location $l \in L$, and each valuation σ, the set $\longrightarrow$ contains exactly one symbolic transition such that $\sigma(g[\overline{d}/\overline{p}])$ is true.

Example 1. Model a simple booking system

5 Inference Using Abstraction

Let us consider the problem of extending finite-state automata learning (as realized. e.g., by th L^* alglorithm) to the learning of infinite state automata, as represented by Symbolic Mealy machines. More precisely, given a *SUT*, whose behavior can be modeled as an SMM $\mathcal{SM}$, we should describe how a component, called the *Learner*, which communicates with the *SUT*, can infer an SMM equivalent to $\mathcal{SM}$ by query learning. In this setup, the *Learner* initially knows the static interface of $\mathcal{SM}$, i.e., the sets I and O of input and output actions together with their arities. It may then ask a sequence of *membership queries*; each one supplying a chosen input word $u \in (\Sigma_I)^*$ and observing the response $\lambda_{\mathcal{SM}}(u)$. After a “sufficient” number of membership membership queries the *Learner* can build a “stable” hypothesis $\mathcal{H}$ from the obtained information. The hypothesis $\mathcal{H}$ should of course agree with $\mathcal{SM}$ on the performed membership queries (i.e., $\lambda_{\mathcal{SM}}(u) = \lambda_{\mathcal{H}}(u)$ whenever u was supplied in a membership query), but must make suitable generalizations for other input words.

The L^* algorithm is designed for finite-state Mealy machines and cannot construct infinite-state models. In order to use it for inferring models of large or infinite-state SMMs, we must somehow transform the behavior of an SMM so that it becomes the behavior of some finite-state Mealy machine. In this section, we present an approach, which has been elaborated in the work by Aarts, Jonsson, Uijen, and Vaandrager [1, 2]. The approach adapts ideas from predicate abstraction [26, 9], which has been successful for extending finite-state model checking to large and infinite state spaces.

In order to introduce our ideas, consider an SMM $\mathcal{SM} = \langle I, O, L, l_0, X, \longrightarrow \rangle$ for which the sets Σ_I and Σ_O of input and output symbols, and the set of valuations of X may be large or even infinite. To apply regular inference to $\mathcal{SM}$, we here propose to define an abstraction from Σ_I and Σ_O to (small) finite sets of *abstract* input and output symbols. The overall idea can be schematically depicted as in Figure 3. The abstraction $\mathcal{A}$ interacts with the *SUT* using the alphabets Σ_I and Σ_O; it interacts with the *Learner* using finite alphabets of symbols $\Sigma_I^{\mathcal{A}}$ and $\Sigma_O^{\mathcal{A}}$. Thus, $\mathcal{A}$ can transform sequences of symbols in Σ_I or Σ_O to sequences of abstract symbols in $\Sigma_I^{\mathcal{A}}$ or $\Sigma_O^{\mathcal{A}}$. If $\mathcal{A}$ is suitable defined, it can also the (possibly) infinite-state behavior of *SUT* into a behavior which can be represented by a finite-state Mealy machine. The *Learner* can then use standard techniques to learn this Mealy machine. Having done this, we can finally

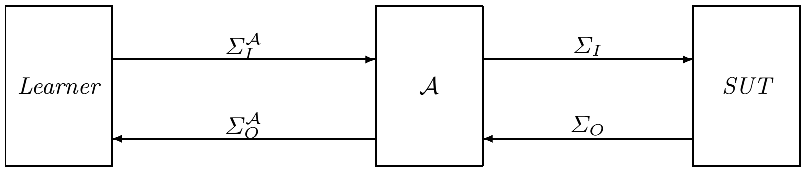


Fig. 3. Introducing an abstraction $\mathcal{A}$ between the *SUT* and the *Learner*

"reverse" the effect of the abstraction $\mathcal{A}$ to obtain an SMM which is equivalent to the original *SUT*.

As a concrete example, in the SMM in our running example, symbols of form getSeats(s) in which the parameter s belongs to the large domain of session identifiers, can be abstracted to symbols of form getSeats(S), where S is a value from a small domain. A natural choice for such a small domain could be the set $\{CUR, BAD\}$, where the value *CUR* denotes that s is the "current" session identifier (supplied in the relevant openSession interaction), and the value *BAD* denotes that s has some other value. Thus, the abstraction of a symbol, such as getSeats(s), in general depends on the previous history of symbols. In model checking using abstraction [26, 9], this dependency is taken into account by letting the abstraction depend on internal state variables. For instance, the *SUT* may have a state variable to remember the "current" session identifier; a predicate abstraction will then only represent whether this internal state variable is equal to the session identifier that is received in the input symbol that is being processed. However, automata learning is performed in a black-box setting where the state variables of the SMM are not accessible. Therefore, these state variables must be recreated in the the abstraction, and be updated to record relevant history information. In our example, the recreated state variables can be *cur_session* and *cur_seats*, where *cur_session* is assigned the session identifier supplied to the *SUT* in the relevant openSession interaction, and *cur_seats* is returned by the *SUT* in response to the relevant getSeats interaction. Typically, these additional state variables must be defined by a user who has some insight into the functioning of the *SUT*. In general, this is a nontrivial task, but in Section 7 we discuss approaches for systematically constructing abstractions for situations where the operation on data is not overly complex.

We can define an abstraction formally as follows.

Definition 4. Let I and O be disjoint finite sets of (input and output) actions. An $\langle I, O\rangle$-*abstraction* is a tuple $\mathcal{A} = \langle \Sigma_I^{\mathcal{A}}, \Sigma_O^{\mathcal{A}}, R, r_0, abstr_I, abstr_O, \delta^{\mathcal{R}}\rangle$, where

– $\Sigma_I^{\mathcal{A}}$ and $\Sigma_O^{\mathcal{A}}$ are finite sets of *abstract* input and output symbols,
– R is a (possibly infinite) set of *local states*,
– $r_0 \in R$ is an *initial local state*,
– $abstr_I : R \times \Sigma_I \mapsto \Sigma_I^{\mathcal{A}}$ maps input symbols of the *SUT* to abstract input symbols,
– $abstr_O : R \times \Sigma_O \mapsto \Sigma_O^{\mathcal{A}}$ maps output symbols of the *SUT* to abstract output symbols, and

– $\delta^{\mathcal{R}} : R \times (\Sigma_I \cup \Sigma_O) \mapsto R$ updates the local state when a new input or output symbol occurs. □

Intuitively, an abstraction $\mathcal{A}$ maps input and output symbols of the *SUT* to abstract input and output symbols, and updates its local state immediately after the occurrence of each symbol.

Let us, as was done for Mealy machines, extend the definitions of the abstraction functions to sequences of input and output symbols. We will use u to range over sequences of input symbols, v to range over sequences of output symbols, and w to range over sequences of pairs (of form a/b) of input and output symbols. Since a Mealy machine interacts with both an input symbol and an output symbol at each transition, we extend the definitions of $\delta^{\mathcal{R}}$, $abstr_I$, and $abstr_O$ by defining:

$$\begin{aligned} \delta^{\mathcal{R}}(r, a/b) &= \delta^{\mathcal{R}}(\delta^{\mathcal{R}}(r, a), b) \\ abstr(r, a/b) &= abstr_I(r, a)/abstr_O(\delta^{\mathcal{R}}(r, a), b) \end{aligned}$$

where a/b is a pair of input and output symbol, and *abstr* maps pairs of input and output symbols to corresponding abstract ones. In the last formula, the abstraction of the input symbol a is performed in the local state r, and the abstraction of the output symbol b is performed wrp. to the local state $\delta^{\mathcal{R}}(r, a)$ reached after having processed the input symbol a.

We thereafter extend $\delta^{\mathcal{R}}$ to sequences of pairs of input and output symbols, by

$$\delta^{\mathcal{R}}(r, \varepsilon) = r \qquad\qquad \delta^{\mathcal{R}}(r, w\ a/b) = \delta^{\mathcal{R}}(\delta^{\mathcal{R}}(r, w), a/b)$$

We can similarly extend the mapping *abstr* from pairs of input and output symbols to sequences of such pairs.

$$\begin{aligned} abstr(r, \varepsilon) &= \varepsilon \\ abstr(r, w\ a/b) &= abstr(r, w)\ abstr(\delta^{\mathcal{R}}(r, w), a/b) \ , \end{aligned}$$

In particular, $abstr(r_0, w)$ is the abstraction of an arbitrary sequence w of input-output pairs.

In a concrete setup for learning a model of the *SUT*, we envisage that the abstraction is performed by introducing a *Mapper* module between the *Learner* and the *SUT*, which carries out the transformations of the abstraction. The *Learner* can then interact with the combination of the *Mapper* and the *SUT*, using the finite sets $\Sigma_I^{\mathcal{A}}$ and $\Sigma_O^{\mathcal{A}}$, whereas the *Mapper* and the *SUT* interact using the alphabets Σ_I and Σ_O. The *Mapper* maintains the local state r of the abstraction. Note that the *Mapper* must transform between original and abstract symbols in two different directions, depending on whether the symbol is an input or output symbol. Each abstract input symbol $a^{\mathcal{A}}$ supplied by the *Learner* is translated by the *Mapper* to a concrete input symbol a such that $a^{\mathcal{A}} = abstr_I(r, a)$, and sent to the *SUT*, while also updating the local state r to $\delta^{\mathcal{R}}(r, a)$. The corresponding reply b by *SUT* is translated to the abstract symbol $abstr_O(\delta^{\mathcal{R}}(r, a), b)$ and sent back to the *Learner*. Finally the local state r is updated to $\delta^{\mathcal{R}}(r, a/b)$.

An example of a possible round of exchanged symbols is depicted in Figure 4. In this round, the abstract symbol $\mathsf{openSession}(\mathrm{USR}, \mathrm{OK})$ is received by the *Mapper*. Here the combination USR, OK represents a valid combination of user and password. The *Mapper* chooses appropriate concrete data values as parameters, including to choose a session identifier to create an input symbol for the *SUT*. It also stores the chosen session identifier into a local variable. The *SUT* recognizes the input symbol as a valid start of a session, and so acknowledges this by returning the provided session identifier. The *Mapper* compares the session identifier returned with its stored local variable containing the session identifier in the preceding input symbol, finds out that they are equal, and therefore transforms 42 into the abstract symbol *CUR*, representing "current session identifier".

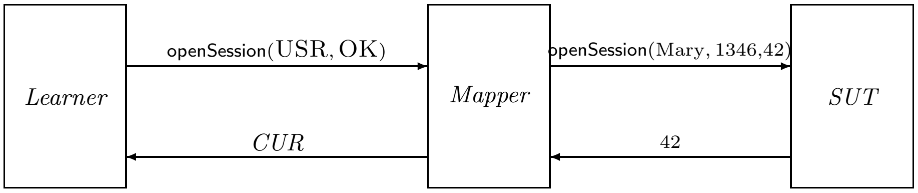


Fig. 4. Introduction of *Mapper* module

Example. Let us suggest an abstraction that could be applied in order to learn a model of the seat booking service. In Section 4, we already described the domains and predicates for modeling data parameters. Let us first suggest a representation of the local state of the abstraction. This can be represented by two state variables:

– *cur_session* which stores the value of the "current" session, and and
– *cur_seats* which stores the set of seats that has been proposed by the service.

The initial values of both variables is $\perp$ (undefined). Thus, a state of the abstraction is a valuation ρ which maps *cur_session* and *cur_seats* to values. Initially, ρ maps these variables to the undefined value.

Let us then define the set of abstract input symbols and the state-dependent mapping $abstr_I$ from input symbols of the service to abstract input symbols. The abstraction of an input symbol depends on whether certain guards, that may be evaluated when such an input symbol is received, hold. For each combination of input symbol and applicable guard, we create a suitable abstract input symbol. In Table 1, we show the different combinations of input symbols and guards, and the corresponding abstract input symbols. For instance, if $\rho(cur\_session) = 42$, and $\rho(cur\_seats) = \{C, D, G\}$, then $abstr_I(\rho, \mathsf{getSeat}(42, F))$ is $\mathsf{getSeat}(CUR, NO\_SEAT)$.

Let us next define the set of abstract output symbols and the mapping $abstr_O$. As described in Section 4, the set of output symbols correspond to data values in domains SESSION, SEATS, and SEAT. In addition, there is an output symbol

Table 1. Mapping from combinations of input symbols and guards to abstract input symbols

input	**guard**	**abstract symbol**
$\mathsf{openSession}(u, p, s)$	$\text{has\_passwd}(u, p)$	$\mathsf{openSession}(\text{USR}, \text{OK})$
	$\neg\text{has\_passwd}(u, p)$	$\mathsf{openSession}(\text{USR}, \text{NOK})$
$\mathsf{getSeats}(s)$	$s = \mathit{cur\_session}$	$\mathsf{getSeats}(\mathit{CUR})$
	$s \neq \mathit{cur\_session}$	$\mathsf{getSeats}(\mathit{BAD})$
$\mathsf{getSeat}(s, \mathit{seat})$	$s = \mathit{cur\_session} \wedge \mathit{seat} \in \mathit{cur\_seats}$	$\mathsf{getSeat}(\mathit{CUR}, \mathit{SEAT})$
	$s = \mathit{cur\_session} \wedge \mathit{seat} \notin \mathit{cur\_seats}$	$\mathsf{getSeat}(\mathit{CUR}, \mathit{NO\_SEAT})$
	$s \neq \mathit{cur\_session} \wedge \mathit{seat} \in \mathit{cur\_seats}$	$\mathsf{getSeat}(\mathit{BAD}, \mathit{SEAT})$
	$s \neq \mathit{cur\_session} \wedge \mathit{seat} \notin \mathit{cur\_seats}$	$\mathsf{getSeat}(\mathit{BAD}, \mathit{NO\_SEAT})$

error which is returned on input that does not make the current session progress. These output symbols are mapped to abstract output symbols as follows.

– Data values in $\mathsf{SESSION}$ are mapped to *CUR* or *BAD*, depending on whether they are equal to *cur_session* or not.
– Data values in SEATS are mapped to *OFFERED* or *NOT_OFFERED*, depending on whether they are equal to the set of seats offered by the service. Here, we have performed a modeling trick in order to be able to model the service as a (deterministic) Mealy machine, in spite of the fact that the set of seats it may return cannot be predicted from the past sequence of input and output symbol. In order not to have to model the return of a set of seats using nondeterminism, we invent a constant, named *offered* (say), which represents the set of seats offered by the service in the session considered.
– Data values in SEAT are mapped to *SEAT* or *NO_SEAT*, depending on whether the seat is a member of the set *cur_seats* or not.
– The output symbol *error* is left unchanged by the abstraction.

Let us finally consider how the state of the abstraction is updated on the occurrence of an input or output symbol. This state is unchanged, except for the following cases.

– When an input symbol of form $\mathsf{openSession}(u, p, s)$ is received, such that $\text{has\_passwd}(u, p)$ and when *cur_session* is previously undefined, then *cur_session* is assigned the value s received in the input symbol.
– When an output symbol in domain SEATS is produced in a situation where *cur_seats* is previously undefined, then *cur_seats* is assigned the value of the output symbol. □

In order to better understand what behavior is obtained by wrapping the *SUT* with the *Mapper*, and which is observed by the *Learner*, let us model the behavior of the combination of the *Mapper* and *SUT*, which we denote by $\mathcal{A}\langle\langle\mathcal{SM}\rangle\rangle$. Unfortunately, $\mathcal{A}\langle\langle\mathcal{SM}\rangle\rangle$ cannot in general be modeled as a (deterministic) Mealy machine. The reason is that each (abstract) input symbol $a^{\mathcal{A}}$ can be translated by the *Mapper* (in state r) to any input symbol a with $a^{\mathcal{A}} = \mathit{abstr}_I(r, a)$: different choices of a will, in general, cause the *SUT* to move to different states and subsequently cause different (abstract) output symbols to be generated. In addition, the *Mapper* should have a defined reaction for the case that there is no input symbol a with $a^{\mathcal{A}} = abstr_I(r, a)$. We therefore need to introduce a generalization of Mealy machines that allows nondeterminism, and represent the behavior of $\mathcal{A}\langle\langle\mathcal{SM}\rangle\rangle$ as a nondeterministic Mealy machine. A nondeterministic Mealy machine differs from a Mealy machine as defined in Definition 1 in that the reception of an input symbol can result in several possible combinations of output symbols and next states. For this situation, it is more suitable to use only the notation $q \xrightarrow{a/b} q'$ to denote that when the machine is in state q and receives input symbol a, a possible reaction is to emit output symbol b and move to state q'.

Let $\mathcal{M}_{\mathcal{SM}} = \langle \Sigma_I, \Sigma_O, Q, q_0, \delta, \lambda\rangle$ denote the Mealy machine model of $\mathcal{SM}$, let $(\Sigma_O^{\mathcal{A}})^{\top} = \Sigma_O^{\mathcal{A}} \cup \{\top\}$ and $R^{\top} = R \cup \{r_{\top}\}$, where $\top$ is an output symbol denoting that the provided abstract input symbol cannot be translated by the *Mapper*. Then the behavior of $\mathcal{A}\langle\langle\mathcal{SM}\rangle\rangle$ can be modeled as a nondeterministic Mealy machine in which

- $\Sigma_I^{\mathcal{A}}$ and $(\Sigma_O^{\mathcal{A}})^{\top}$ are the sets of input and output symbols,
- $Q \times R^{\top}$ is the set of states,
- $\langle q_0, r_0\rangle$ is the initial state, and
- whenever $\mathcal{A}\langle\langle\mathcal{SM}\rangle\rangle$ is in state $\langle q, r\rangle$ and receives an abstract input symbol $a^{\mathcal{A}}$, then for any concrete input symbol a such that $a^{\mathcal{A}} = abstr_I(r, a)$
  - the state $\langle q, r\rangle$ can be updated to $\langle \delta(q, a), \delta^{\mathcal{R}}(r, a/b)\rangle$, where $b = \lambda(q, a)$ is the output symbol returned by $\mathcal{SM}$, and
  - the abstract output symbol $abstr_O(\delta^{\mathcal{R}}(r, a), b)$ can be produced.

  We denote this possible symbol exchnage by

$$\langle q, r\rangle \xrightarrow{abstr_I(r,a)/abstr_O(\delta^{\mathcal{R}}(r,a),\lambda(q,a))} \langle \delta(q, a), \delta^{\mathcal{R}}(r, a/\lambda(q, a))\rangle \quad .$$

  For the case where there is no concrete input symbol a such that $a^{\mathcal{A}} = abstr_I(r, a)$, the output symbol $\top$ is produced and the state $\langle q, r\rangle$ is updated to $r_{\top}$, where it remains, i.e.,

  - $\langle q, r\rangle \xrightarrow{a^{\mathcal{A}}/\top} \langle q, r_{\top}\rangle$, and
  - $\langle q, r_{\top}\rangle \xrightarrow{a^{\mathcal{A}}/\top} \langle q, r_{\top}\rangle$ for any $a^{\mathcal{A}} \in \Sigma_I^{\mathcal{A}}$.

Although the behavior of $\mathcal{A}\langle\langle\mathcal{SM}\rangle\rangle$ must, in general, be modeled as a Mealy machine, which is internally nondeterministic, it is still possible that its external behavior will appear to the *Learner* as being deterministic. The *Learner* can only observe the sequences of abstract output symbols that are produced in response to provided input sequences. So, for a sequence $a_1^{\mathcal{A}} \cdots a_n^{\mathcal{A}}$ of abstract input symbols, define $\lambda^{\mathcal{A}\langle\langle\mathcal{SM}\rangle\rangle}(\langle q, r\rangle, a_1^{\mathcal{A}} \cdots a_n^{\mathcal{A}})$ as the set of sequences of output symbols of form $b_1^{\mathcal{A}} \cdots b_n^{\mathcal{A}}$ such that for some sequence of states $\langle q_1, r_1\rangle \cdots \langle q_n, r_n\rangle$ we have

$$\langle q, r\rangle \xrightarrow{a_1^{\mathcal{A}}/b_1^{\mathcal{A}}} \langle q_1, r_1\rangle \xrightarrow{a_2^{\mathcal{A}}/b_2^{\mathcal{A}}} \cdots \xrightarrow{a_n^{\mathcal{A}} co/b_n^{\mathcal{A}}} \langle q_n, r_n\rangle$$

Intuitively, $\lambda^{\mathcal{A}\langle\langle \mathcal{SM}\rangle\rangle}(\langle q, r\rangle, a_1^{\mathcal{A}} \cdots a_n^{\mathcal{A}})$ is the set of abstract output sequences that may be generated by $\mathcal{A}\langle\langle \mathcal{SM}\rangle\rangle$ in response to $a_1^{\mathcal{A}} \cdots a_n^{\mathcal{A}}$, starting from state $\langle q, r\rangle$. In particular, $\lambda^{\mathcal{A}\langle\langle \mathcal{SM}\rangle\rangle}(\langle q_0, r_0\rangle, u)$ is the set of sequences that may result from the input sequence u.

If tha input-output behavior of $\mathcal{A}\langle\langle \mathcal{SM}\rangle\rangle$ is equivalent to that of a deterministic Mealy machine, and if this (deterministic) Mealy machine is finite-state, it will be possible to use L^* for learning a model of its external behavior. Indeed, a well-designed abstraction will preserve the determinism of the *SUT* so that the input-output behavior of $\mathcal{A}\langle\langle \mathcal{SM}\rangle\rangle$ behaves deterministically, i.e., each sequence of supplied (abstract) input symbols uniquely determines the subsequently produced (abstract) output symbol.

We make a formal definition of the condition under which the abstraction will present a deterministic view to the *Learner*.

Definition 5. Let $\mathcal{SM} = \langle I, O, L, l_0, X, \longrightarrow\rangle$ be an SMM, and let $\mathcal{A} = \langle \Sigma_I^{\mathcal{A}}, \Sigma_O^{\mathcal{A}}, R, r_0, \mathit{abstr}_I, \mathit{abstr}_O, \delta^{\mathcal{R}}\rangle$ be an $\langle I, O\rangle$-abstraction. Then $\mathcal{A}$ is *adequate* for $\mathcal{SM}$ if for any sequence $u \in (\Sigma_I^{\mathcal{A}})^*$ of abstract input symbols, the set $\lambda^{\mathcal{A}\langle\langle \mathcal{SM}\rangle\rangle}(\langle q_0, r_0\rangle, u)$ of correspondingly generated output sequences has at most one element. □

Intuitively, adequacy means that $\mathcal{A}\langle\langle \mathcal{SM}\rangle\rangle$ exhibits a deterministic mapping from sequences of abstract input symbols received by the *Mapper* to sequences of abstract output symbols produced by the *Mapper* after abstracting the output of the *SUT*. If $\mathcal{A}$ is adequate for $\mathcal{SM}$, then the *Learner* will perceive that $\mathcal{A}\langle\langle \mathcal{SM}\rangle\rangle$ is equivalent to a (deterministic) Mealy machine (which may or may not be finite-state). For any deterministic mapping from sequences of abstract input symbols to sequences of abstract output symbols, there is a minimal Mealy machine which generates it. This Mealy machine can be defined by a Nerode-like quotient construction, as follows.

Let $Q^{\langle \mathcal{SM}, \mathcal{A}\rangle}$ denote the set of states of $\mathcal{A}\langle\langle \mathcal{SM}\rangle\rangle$ that are reachable. More precisely, $Q^{\langle \mathcal{SM}, \mathcal{A}\rangle}$ is the smallest subset of $Q \times R$ which includes $\langle q_0, r_0\rangle$ and such that $\langle q, r\rangle \in Q^{\langle \mathcal{SM}, \mathcal{A}\rangle}$ implies $\langle \delta(q, a), \delta^{\mathcal{R}}(r, a/\lambda(q, a)\rangle \in Q^{\langle \mathcal{SM}, \mathcal{A}\rangle}$ for all $a \in \Sigma_I$). Note that we have excluded states where the *Mapper* has reached $r_\top$.

Define the equivalence $\simeq$ on $Q^{\langle \mathcal{SM}, \mathcal{A}\rangle}$ by $\langle q, r\rangle \simeq \langle q', r'\rangle$ if $\lambda^{\mathcal{A}\langle\langle \mathcal{SM}\rangle\rangle}(\langle q, r\rangle, u) = \lambda^{\mathcal{A}\langle\langle \mathcal{SM}\rangle\rangle}(\langle q', r'\rangle, u)$ for any sequence of abstract input symbols $u \in (\Sigma_I^{\mathcal{A}})^*$. Intuitively, two elements of $Q^{\langle \mathcal{SM}, \mathcal{A}\rangle}$ are equivalent if they cannot be distinguished by the *Learner*, i.e., any two subsequent sequences of input symbols that are identified by abstr_I trigger two subsequent output words that are identified by abstr_O.

If $\mathcal{A}$ is adequate for $\mathcal{SM}$, then the input-output behavior of $\mathcal{A}\langle\langle \mathcal{SM}\rangle\rangle$ is equal to that of a deterministic Mealy machine $\mathcal{M}^{\mathcal{A}}$ (in the sense that $\lambda^{\mathcal{A}\langle\langle \mathcal{SM}\rangle\rangle}(\langle q_0, r_0\rangle, u) = \{\lambda_{\mathcal{M}^{\mathcal{A}}}(u)\}$ for any sequence $u \in (\Sigma_I^{\mathcal{A}})^*$ of abstract input symbols), which is defined by $\mathcal{M}^{\mathcal{A}} = \langle \Sigma_I^{\mathcal{A}}, \Sigma_O^{\mathcal{A}}, Q^{\mathcal{A}}, q_0^{\mathcal{A}}, \delta^{\mathcal{A}}, \lambda^{\mathcal{A}}\rangle$, where

- $Q^{\mathcal{A}} = Q^{\langle \mathcal{SM}, \mathcal{A}\rangle} / \simeq \cup \{q_\top\}$, i.e., the set of states is the set of equivalence classes under $\simeq$ plus an extra state $q_\top$ denoting that an abstract input symbol with no corresponding concrete input symbol has been received,

- $q_0^{\mathcal{A}} = [\langle q_0, r_0 \rangle]_{\simeq}$ is the equivalence class of the initial state of $\mathcal{A}\langle\langle \mathcal{SM} \rangle\rangle$,
- $\delta^{\mathcal{A}}$ and $\lambda^{\mathcal{A}}$ are defined as follows:
  for any $a \in \Sigma_I$ with $abstr_I(r, a) = a^{\mathcal{A}}$ we have
  - $\delta^{\mathcal{A}}([\langle q, r \rangle]_{\simeq}, a^{\mathcal{A}}) = [\langle \delta(q, a), \delta^{\mathcal{R}}(r, a/\lambda(q, a)) \rangle]_{\simeq}$, and
  - $\lambda^{\mathcal{A}}([\langle q, r \rangle]_{\simeq}, a^{\mathcal{A}}) = abstr_O(\delta^{\mathcal{R}}(r, a), \lambda(q, a))$,

  for any $a^{\mathcal{A}} \in \Sigma_I^{\mathcal{A}}$ s.t. there is no $a \in \Sigma_I$ with $abstr_I(r, a) = a^{\mathcal{A}}$ we have
  - $\lambda^{\mathcal{A}}([\langle q, r \rangle]_{\simeq}, a^{\mathcal{A}}) = \top$, and
  - $\delta^{\mathcal{A}}([\langle q, r \rangle]_{\simeq}, a^{\mathcal{A}}) = q_{\top}$.

  To complete the definition, we define $\delta^{\mathcal{A}}(q_{\top}, a^{\mathcal{A}}) = q_{\top}$ for any $a^{\mathcal{A}} \in \Sigma_I^{\mathcal{A}}$.

The definition of $\simeq$ can be used to show that $\mathcal{M}^{\mathcal{A}}$ is well-defined.

The Mealy machine $\mathcal{M}^{\mathcal{A}}$ may be finite- or infinite-state. If a finite-state Mealy machine $\mathcal{M}^{\mathcal{A}} = \langle \Sigma_I^{\mathcal{A}}, \Sigma_O^{\mathcal{A}}, Q^{\mathcal{A}}, q_0^{\mathcal{A}}, \delta^{\mathcal{A}}, \lambda^{\mathcal{A}} \rangle$ is produced by the *Learner*, then we must finally "reverse" the effect of the abstraction $\mathcal{A}$ to obtain the original SMM $\mathcal{SM}$, such that $\mathcal{A}\langle\langle \mathcal{SM} \rangle\rangle$ is equivalent to $\mathcal{M}^{\mathcal{A}}$. In general, there can of course be many SMMs with this property. In order that the SMMs be determined uniquely up to equivalence (which is anyway the best we can hope for), it is necessary that each abstract output symbol correspond to a uniquely determined concrete output symbol generated by the SMM. We formulate this as follows.

Definition 6. An $\langle I, O \rangle$-abstraction $\mathcal{A} = \langle \Sigma_I^{\mathcal{A}}, \Sigma_O^{\mathcal{A}}, R, r_0, abstr_I, abstr_O, \delta^{\mathcal{R}} \rangle$ is *unambiguous* if for all abstract output symbols $b^{\mathcal{A}}$ and all $r \in R$ there is at most one output symbol b such that $b^{\mathcal{A}} = abstr_O(\delta^{\mathcal{R}}(r, a), b)$ for some input symbol $a \in \Sigma_I$. □

Intuitively, this means that we can deduce which output symbol is produced by $\mathcal{SM}$ by seeing only its abstraction.

If $\mathcal{A}$ is unambiguous, and $\mathcal{M}^{\mathcal{A}} = \langle \Sigma_I^{\mathcal{A}}, \Sigma_O^{\mathcal{A}}, Q^{\mathcal{A}}, q_0^{\mathcal{A}}, \delta^{\mathcal{A}}, \lambda^{\mathcal{A}} \rangle$ is a finite-state mealy machine, define $\mathcal{A}^{-1}\langle\langle \mathcal{M}^{\mathcal{A}} \rangle\rangle$ as be the Mealy machine $\langle \Sigma_I, \Sigma_O, Q^{\mathcal{A}} \times R, \langle q_0^{\mathcal{A}}, r_0 \rangle, \delta, \lambda \rangle$, where δ and λ are defined by

- $\lambda(\langle q^{\mathcal{A}}, r \rangle, a) = b$, where b the unique output symbol such that $\lambda^{\mathcal{A}}(q^{\mathcal{A}}, abstr_I(r, a)) = abstr_O(\delta^{\mathcal{R}}(r, a), b)$, and
- $\delta(\langle q^{\mathcal{A}}, r \rangle, a) = \langle \delta^{\mathcal{A}}(q^{\mathcal{A}}, abstr_I(r, a)), \delta^{\mathcal{R}}(r, a/b) \rangle$.

We can now prove that under the conditions we have introduced, the *SUT* can be inferred from $\mathcal{M}^{\mathcal{A}}$, of course up to equivalence.

Proposition 1. *If $\mathcal{A}$ is unambiguous and adequate for $\mathcal{SM}$, and if $\mathcal{A}\langle\langle \mathcal{SM} \rangle\rangle$ is equivalent to $\mathcal{M}^{\mathcal{A}}$ (i.e., $\lambda^{\mathcal{A}\langle\langle \mathcal{SM} \rangle\rangle}(\langle q_0, r_0 \rangle, u) = \{\lambda_{\mathcal{M}^{\mathcal{A}}}(u)\}$ for any $u \in (\Sigma_I^{\mathcal{A}})^*$), then $\mathcal{SM}$ is equivalent to $\mathcal{A}^{-1}\langle\langle \mathcal{M}^{\mathcal{A}} \rangle\rangle$.* □

The proposition can be proven by establishing that $\mathcal{A}^{-1}\langle\langle \mathcal{M}^{\mathcal{A}} \rangle\rangle$ satisfies the conditions of the propsition, i.e., that $\mathcal{A}\langle\langle \mathcal{A}^{-1}\langle\langle \mathcal{M}^{\mathcal{A}} \rangle\rangle \rangle\rangle$ is equivalent to $\mathcal{M}^{\mathcal{A}}$, and by establishing (e.g., by induction on the length of u) that the ouput generated in response to an input sequence u, by any $\mathcal{SM}$ such that $\mathcal{A}\langle\langle \mathcal{SM} \rangle\rangle$ is equivalent to $\mathcal{M}^{\mathcal{A}}$, is uniquely determined.

6 Illustrating Example

In this section, we sketch how a model of the booking service could be obtained by combining automata learning, and the abstraction that was developed in the course of the two previous sections.

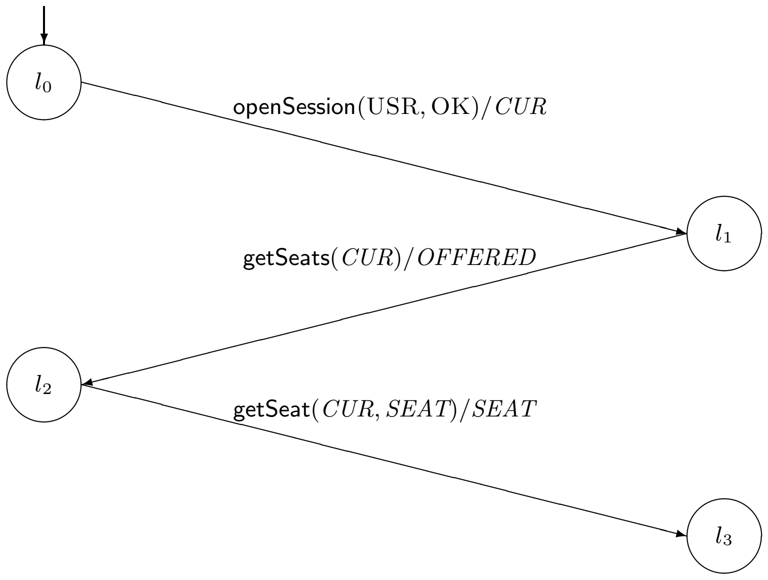

l_0
$\mathsf{openSession}(\mathrm{USR}, \mathrm{OK})/\mathit{CUR}$
l_1
$\mathsf{getSeats}(\mathit{CUR})/\mathit{OFFERED}$
l_2
$\mathsf{getSeat}(\mathit{CUR}, \mathit{SEAT})/\mathit{SEAT}$
l_3

Fig. 5. Learned Abstract Mealy machine (self-loops suppressed)

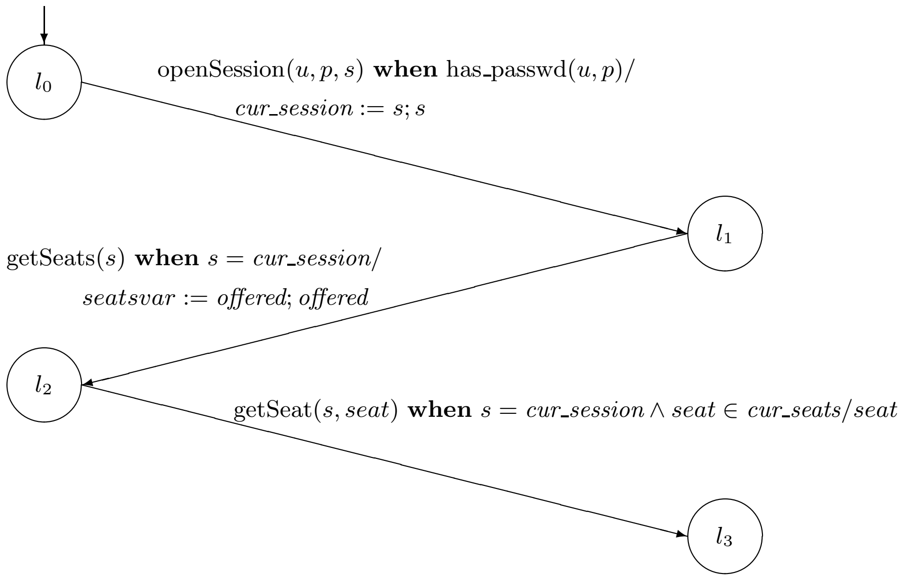

l_0
$\mathrm{openSession}(u, p, s)$ **when** $\mathrm{has\_passwd}(u, p)/$
$\mathit{cur\_session} := s; s$
l_1
$\mathrm{getSeats}(s)$ **when** $s = \mathit{cur\_session}/$
$\mathit{seatsvar} := \mathit{offered}; \mathit{offered}$
l_2
$\mathrm{getSeat}(s, \mathit{seat})$ **when** $s = \mathit{cur\_session} \wedge \mathit{seat} \in \mathit{cur\_seats}/\mathit{seat}$
l_3

Fig. 6. Constructed Symbolic Mealy machine (self-loops suppressed)

Having supplied the abstraction described in the previous section, we can now employ the L^* algorithm to learn a finite-state Mealy machine, which interacts using the sets $\Sigma_I^{\mathcal{A}}$ and $\Sigma_O^{\mathcal{A}}$ of symbols. Assume that the result is as described by the finite-state Mealy machine in Figure 5. In this figure, we have omitted all transitions that return *error*, and concentrate on those that make the session progress.

Starting from the finite-state Mealy machine in Figure 5, we can apply the construction described at the end of the preceding section to generate a possible SMM that models the service. It is shown in Figure 6.

7 Systematic Construction of Abstractions

The construction of a suitable abstraction is crucial for successful inference of an SMM $\mathcal{SM}$. In this subsection, we discuss how a successful abstraction can be constructued more systematically. A necessary prerequisite for constructing an abstraction is obviously that the sets I and O of input and output actions of $\mathcal{SM}$, together with their arities, are known *a priori.*

Furthermore, it appears necessary to have some *a priori* knowledge about how $\mathcal{SM}$ stores and manipulates data that it receives and emits. On the other hand, the control aspects of $\mathcal{SM}$ can be inferred by the *Learner* using automata learning, provided that the abstraction is "good enough". In the following, we give some sufficient criteria for "good enough" abstractions, under which learning and $\mathcal{SM}$, according to the technique described in Section 5 can be successful. At the end of this section, we discuss how such knowledge can sometimes be obtained by testing and experimentation on $\mathcal{SM}$.

In the running example in the previous subsection, we see that the abstraction mapping for input symbols uses expressions that become guards in the resulting SMM, and that the abstraction mapping for output symbols uses expressions that occur in output expressions of the SMM. When $\mathcal{SM}$ is only available as a black box, such an abstraction can be produced if the following information is available.

– An "overestimate" of the information that is stored in state variables of $\mathcal{SM}$. More precisely, such an overestimate can be represented by a set R of states of the abstraction and an update function $\delta^{\mathcal{R}}$ such that after any sequence of pairs of input and output symbols, the information in the "data state", represented by the current valuation σ of state variables, of $\mathcal{SM}$ can be obtained from the current state r of the abstraction. One way to formalize is by finding a mapping h from the set of valuations of the state variables X to the set r of states of the *Mapper*, which has the properties that
  - $h(\sigma_0) = r_0$, and
  - for any symbolic transition in $\longrightarrow$ of form

$$\begin{tikzcd} l \arrow[r, "\alpha(\overline{p})\ \mathbf{when}\ g\ /\ x_1,\ldots,x_k := e_1,\ldots,e_k\ ;\ \beta({e^{out}}_1,\ldots,{e^{out}}_m)"] & l' \end{tikzcd}$$

  and valuation σ and data values $\overline{d}$ such that $\sigma(g[\overline{d}/\overline{p}])$ evaluates to true, then if $h(\sigma) = r$, then it holds that for σ' defined by
    * $\sigma'(x_i) = \sigma(e_i[\overline{d}/\overline{p}])$ for $1 \le i \le k$, and
    * $\sigma'(x) = \sigma(x)$ if x is not among $x_1, \ldots, x_k$,

we have $h(\sigma') = \delta^{\mathcal{R}}(r, \alpha(\overline{d})/\beta(\sigma'(e_1^{out}[\overline{d}/\overline{p}]), \ldots, \sigma'(e_m^{out}[\overline{d}/\overline{p}])))$.

– The abstraction should distinguish between the different symbolic transitions from a location. More precisely, this means that if from a location and from some state $\langle l, \sigma \rangle$ with $h(\sigma) = r$, there are two different symbolic transitions taken for input $\alpha(\overline{d})$ and $\alpha'(\overline{d}')$, then $abstr_I(r, \alpha(\overline{d})) \neq abstr_I(r, \alpha'(\overline{d}'))$. This can be achieved by letting each possible guard correspond to a different abstract input symbol. For the case that the abstraction is not fine enough to distinguish between symbolic transitions that cause different output, a technique for refining the abstraction on-the-fly, during the learning process, has been developed by Howar, Steffen, and Merten [19].
– The abstraction should be unambiguous. This can be achieved if different output expressions are mapped to different abstract output symbols. For instance, one could let abstract output symbols "be" the output expressions that can occur in symbolic transitions, assuming that an output expression is uniquely obtainable from the actual output symbol produced.

Under the above assumptions, we can construct an abstraction which maps combinations of parameterized input actions and guards in a possibe SMM to abstract input symbols, and maps combinations of expressions in output symbols of a possible SMM to abstract output symbols, as in the running example. The updates to state variables will simply consist in assigning some input parameters to state variables: the problem here is to decide which input parameters will influence the future behavior of $\mathcal{SM}$, and must be remembered in state variables. In our experiments, we have made this decision based on observing the response of $\mathcal{SM}$ to selected input strings, i.e., by posing membership queries, and saving those parameter values that are used to produce future output. For parameter values on which the only performed operation is a test for equality, such as the *id* parameter of the running example, we have made these ideas more precise in our earlier work [6], as follows:

Consider an input string u, which contains a parameter value d. We observe the output of $\mathcal{M}$ in response to u and to selected continuations of u, and decide to store d in a state variable if there is some continuation v of u such that d is used to produce the response to v. More precisely, this happens if there is a fresh (i.e., previously unused) data value d' such that the response $\lambda(\delta(q_0, u), v)$ to v and the response $\lambda(\delta(q_0, u), v[d'/d])$ to $v[d'/d]$ (i.e., v where all occurrences of d have been replaced by d') satisfy $\lambda(\delta(q_0, u), v)[d'/d] \neq \lambda(\delta(q_0, u), v[d'/d])$, i.e., $\mathcal{SM}$ does not treat d in the same way as a fresh (previously unused) value d'. This happens, e.g., if $\lambda(\delta(q_0, u), v[d'/d])$ contains the data value d implying that d must have been remembered before seeing the subsequent input $v[d'/d]$, and that d should be stored in a state variable.

8 Conclusions and Future Work

We have considered the problem of extending automata learning to incorporate data parameters, including their influence on control behavior. We concentrated on presenting an approach that adapts ideas using abstraction that have been successfully applied in formal verification. This approach has been used on some nontrivial examples [1, 2], and techniques for revising abstractions by need have been developed [19]. However, it is clear that much work remains in order to make automata learning with data easily applicable to a wide class of systems. Issues that need to be addressed include to remove (some of) the need for manual construction of abstractions: this could be addressed by developing more canonical models for automata with data. Another issue is that the determinism of the Mealy machine model is limiting the modeling power: ways should be found to effectively learn nondeterministic models.

Acknowledgment. This paper builds on joint work with several present and former collaborators, including Fides Aarts, Therese Bohlin, Sofia Cassel, Olga Grinchtein, Falk Howar, Maik Merten, Bernhard Steffen, Johan Uijen, and Frits Vaandrager. Mistakes and inconsistencies are caused by the author.

References

1. Aarts, F., Jonsson, B., Uijen, J.: Generating models of infinite-state communication protocols using regular inference with abstraction. In: Petrenko, A., Simão, A., Maldonado, J.C. (eds.) ICTSS 2010. LNCS, vol. 6435, pp. 188–204. Springer, Heidelberg (2010)
2. Aarts, F., Schmaltz, J., Vaandrager, F.: Inference and abstraction of the biometric passport. In: Margaria, T., Steffen, B. (eds.) ISoLA 2010. LNCS, vol. 6415, pp. 673–686. Springer, Heidelberg (2010)
3. Ammons, G., Bodik, R., Larus, J.: Mining specifications. In: Proc. 29th ACM Symp. on Principles of Programming Languages, pp. 4–16 (2002)
4. Angluin, D.: Learning regular sets from queries and counterexamples. Information and Computation 75(2), 87–106 (1987)
5. Berg, T., Jonsson, B., Raffelt, H.: Regular inference for state machines with parameters. In: Baresi, L., Heckel, R. (eds.) FASE 2006. LNCS, vol. 3922, pp. 107–121. Springer, Heidelberg (2006)
6. Berg, T., Jonsson, B., Raffelt, H.: Regular inference for state machines using domains with equality tests. In: Fiadeiro, J.L., Inverardi, P. (eds.) FASE 2008. LNCS, vol. 4961, pp. 317–331. Springer, Heidelberg (2008)
7. Broy, M., Jonsson, B., Katoen, J.-P., Leucker, M., Pretschner, A. (eds.): Model-Based Testing of Reactive Systems. LNCS, vol. 3472. Springer, Heidelberg (2005)
8. Brun, Y., Ernst, M.: Finding latent code errors via machine learning over program executions. In: ICSE 2004: 26th Int. Conf. on Software Enginering (May 2004)
9. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. Journal of the ACM 50(5), 752–794 (2003)
10. Cobleigh, J.M., Giannakopoulou, D., Păsăreanu, C.S.: Learning assumptions for compositional verification. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 331–346. Springer, Heidelberg (2003)
11. Dupont, P.: Incremental regular inference. In: Miclet, L., de la Higuera, C. (eds.) ICGI 1996. LNCS, vol. 1147, pp. 222–237. Springer, Heidelberg (1996)

12. Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The daikon system for dynamic detection of likely invariants. Sci. Comput. Program. 69(1-3), 35–45 (2007)
13. Gold, E.M.: Language identification in the limit. Information and Control 10(5), 447–474 (1967)
14. Grinchtein, O.: Learning of Timed Systems. PhD thesis, Dept. of IT, Uppsala University, Sweden (2008)
15. Grinchtein, O., Jonsson, B., Leucker, M.: Learning of event-recording automata. In: Lakhnech, Y., Yovine, S. (eds.) FORMATS 2004 and FTRTFT 2004. LNCS, vol. 3253, pp. 379–395. Springer, Heidelberg (2004)
16. Groce, A., Peled, D.A., Yannakakis, M.: Adaptive model checking. In: Katoen, J.-P., Stevens, P. (eds.) TACAS 2002. LNCS, vol. 2280, pp. 357–370. Springer, Heidelberg (2002)
17. Groz, R., Li, K., Petrenko, A., Shahbaz, M.: Modular system verification by inference, testing and reachability analysis. In: Suzuki, K., Higashino, T., Ulrich, A., Hasegawa, T. (eds.) TestCom/FATES 2008. LNCS, vol. 5047, pp. 216–233. Springer, Heidelberg (2008)
18. Hagerer, A., Hungar, H., Niese, O., Steffen, B.: Model generation by moderated regular extrapolation. In: Kutsche, R.-D., Weber, H. (eds.) FASE 2002. LNCS, vol. 2306, pp. 80–95. Springer, Heidelberg (2002)
19. Howar, F., Steffen, B., Merten, M.: Automata learning with automated alphabet abstraction refinement. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 263–277. Springer, Heidelberg (2011)
20. Huima, A.: Implementing conformiq qtronic. In: Petrenko, A., Veanes, M., Tretmans, J., Grieskamp, W. (eds.) TestCom/FATES 2007. LNCS, vol. 4581, pp. 1–12. Springer, Heidelberg (2007)
21. Hungar, H., Niese, O., Steffen, B.: Domain-specific optimization in automata learning. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 315–327. Springer, Heidelberg (2003)
22. Issarny, V., Steffen, B., Jonsson, B., Blair, G.S., Grace, P., Kwiatkowska, M.Z., Calinescu, R., Inverardi, P., Tivoli, M., Bertolino, A., Sabetta, A.: Connect challenges: Towards emergent connectors for eternal networked systems. In: ICECCS, pp. 154–161 (2009)
23. Kearns, M., Vazirani, U.: An Introduction to Computational Learning Theory. MIT Press, Cambridge (1994)
24. Li, K., Groz, R., Shahbaz, M.: Integration testing of distributed components based on learning parameterized I/O models. In: Najm, E., Pradat-Peyre, J.-F., Donzeau-Gouge, V.V. (eds.) FORTE 2006. LNCS, vol. 4229, pp. 436–450. Springer, Heidelberg (2006)
25. Lo, D., Maoz, S.: Scenario-based and value-based specification mining: better together. In: ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, pp. 387–396. ACM, New York (2010)
26. Loiseaux, C., Graf, S., Sifakis, J., Boujjani, A., Bensalem, S.: Property preserving abstractions for the verification of concurrent systems. Formal Methods in System Design 6(1), 11–44 (1995)
27. Lorenzoli, D., Mariani, L., Pezzè, M.: Automatic generation of software behavioral models. In: Proc. ICSE 2008: 30th Int. Conf. on Software Enginering, pp. 501–510 (2008)
28. Mariani, L., Pezzè, M.: Dynamic detection of COTS components incompatibility. IEEE Software 24(5), 76–85 (2007)

29. Peled, D., Vardi, M.Y., Yannakakis, M.: Black box checking. In: Wu, J., Chanson, S.T., Gao, Q. (eds.) Formal Methods for Protocol Engineering and Distributed Systems, FORTE/PSTV, pp. 225–240. Kluwer, Beijing (1999)
30. Rivest, R., Schapire, R.: Inference of finite automata using homing sequences. Information and Computation 103, 299–347 (1993)
31. Shahbaz, M., Li, K., Groz, R.: Learning and integration of parameterized components through testing. In: Petrenko, A., Veanes, M., Tretmans, J., Grieskamp, W. (eds.) TestCom/FATES 2007. LNCS, vol. 4581, pp. 319–334. Springer, Heidelberg (2007)
32. Shu, G., Lee, D.: Testing security properties of protocol implementations - a machine learning based approach. In: Proc. ICDCS 2007, 27th IEEE Int. Conf. on Distributed Computing Systems, Toronto, Ontario. IEEE Computer Society, Los Alamitos (2007)
33. Trakhtenbrot, B., Barzdin, J.: Finite automata: behaviour and synthesis. North-Holland, Amsterdam (1973)

Dependability and Performance Assessment of Dynamic CONNECTed Systems

Antonia Bertolino, Antonello Calabró,
Felicita Di Giandomenico, and Nicola Nostro

Istituto di Scienza e Tecnologie dell'Informazione,
Consiglio Nazionale delle Ricerche
via Moruzzi 1, I-56124, Italy
name.surname@isti.cnr.it

Abstract. In this chapter we present approaches for analysis and monitoring of dependability and performance of CONNECTed systems, and their combined usage. These approaches need to account for dynamicity and evolvability of CONNECTed systems. In particular, the chapter covers the quantitative assessment of dependability and performance properties through a stochastic model-based approach: first an overview of dependability-related measurements and stochastic model-based approaches provides the necessary background. Then, our proposal in CONNECT of an automated and modular dependability analysis framework for dynamically CONNECTed systems is described. This framework can be used off-line for system design (specifically, in CONNECT, for CONNECTor synthesis), and on-line, to continuously assess system behaviour and detect possible issues arising at run-time. For the latter purpose, a generic, flexible and modular monitoring infrastructure has been developed. Monitoring is at the core of the CONNECT vision, in order to ensure run-time observation of specified quantitative properties and possibly trigger adequate reactions. We focus here on the interaction chain between monitoring and analysis, to allow for on-line continuous validation of specified dependability and performance properties. Illustrative examples of applications of analysis and monitoring are provided with reference to the CONNECT Terrorist Alert scenario.

1 Introduction

Modern software applications are more and more conceived as dynamically adaptable and evolvable sets of components that must be able to modify their behaviour at run-time to tackle the continuous changes in the unpredictable open-world settings [BGD06]. On the other hand, these systems are increasingly pervasive and their improper behaviour will produce effects on our everyday lives and business, which can range from annoying ones up to sometime even critical consequences. Therefore, we need to ensure that these dynamic systems provide the required non-functional properties, such as reliability, availability, performance, security and trust, and so on, and continue to do so even after evolution and adaptation.

In such partially unknown and evolving contexts, dependability analysis [Lap95] calls for on-line support to enhance the accuracy of preliminary estimates performed at design time. Indeed, as we entrust more and more responsibilities to distributed software systems, the need arises to augment them with powerful oversight and management functions in order to allow continuous and flexible monitoring of their behaviour.

In this chapter we tackle the challenge of dependability and performance analysis in dynamic CONNECTed systems. We present our preliminary results obtained in the context of the European Project CONNECT [CON13], which considers dynamic environments populated by heterogeneous Networked Systems within disconnected isles, willing to communicate with each other despite differing and evolving technologies.

In CONNECT, communication between heterogeneous systems is seamlessly supported by CONNECT Enablers, which make on-the-fly interoperation possible by synthesising and deploying mediating software bridges, called CONNECTors. Specifically, the main Enablers in the CONNECT architecture include (please refer to [GGB+11] for a complete description): *Discovery*, which discovers mutually interested Networked Systems (NSs), and retrieves information on the NS interfaces; *Learning*, which possibly completes the specifications of the NSs through a learning procedure; *Synthesis* and *Deployment*, which perform, respectively, the dynamic synthesis of the mediating CONNECTors and their deployment; *Dependability&Performance*, which uses a model-based analysis to support *Synthesis* in the definition of a dependable CONNECTor; *Security&Trust*, which assess and enforce security, privacy and trust aspects, and finally *Monitoring*, which continuously monitors the deployed CONNECTor to update the CONNECTor specification used by the other Enablers with run-time data.

To accomplish dependability and performance analysis in such a complex and evolving context, both off-line and on-line approaches are pursued, to cover a wider range of needs. As commonly intended in the literature, off-line analysis refers to activities devoted to analyse the system at hand before deployment, or even after its deployment but in isolation with respect to the system in operation. On the contrary, on-line analysis refers to activities performed while the system is in operation, so accounting for the detailed system and environment aspects during that specific system execution. We adopt the off-line and on-line terms with such meanings.

Analysis at the early stage of a development process is of paramount importance to achieve the required functional and non-functional properties. In fact, early evaluation of the concepts and architectural choices prevents wasting time and resources by promptly identifying possible design deficiencies or helping in performing design decisions by comparing different alternative architectural solutions and selecting the most suitable one.

Nevertheless, the incomplete a priori knowledge about the operating system and environment unavoidably undermines the accuracy of the considered elements and, hence, of the analysis results. In this perspective, monitoring becomes a key technological enabler for dependability assurance, as it provides the enabling infrastructure to prolong software lifecycle after deployment, by supporting run-time verification and on-line adaptation.

Therefore, in Connect, both an automated approach to off-line dependability analysis adopting model-based analysis, to support the design of dependable connectors, and event-based monitoring, to support dependability and performance run-time analysis, are under definition and development. In this chapter, the two approaches are first individually presented, pointing out their respective architectures and their role in the Connect framework. Then, we focus on their synergic use, to allow refining model-based dependability and performance analysis in distributed dynamic systems through monitoring.

The continuous interplay and refinement between model-based analysis and run-time monitoring is today emerging as an irremissible direction of software development. A software module (the whole software system or part of it) is repeatedly analysed through model-based analysis and refined in its design until it proves to satisfy specified non-functional quantitative requirements. Once such a proper design is obtained, it is deployed in a suitable computing environment and put in operation. At run-time, the deployed software system must be monitored to be sure that its execution respects the required properties. Data collected through monitoring constitute invaluable information to be exploited for: i) validating the models generated through model-based software engineering sub-process, and ii) continuously refining the analysis by overcoming the possible inaccuracy in the values of the model parameters due to incomplete knowledge or to evolution of the elements involved in the analysis.

The rest of the chapter is structured as follows. Section 2 presents some background material about dependability and performance properties and related analysis approaches. The approaches undertaken in Connect to perform dependability and performance assessment of Connected systems are dealt with in Section 3, namely model-based analysis and event-based monitoring. In addition to presenting them individually, emphasis is put on their synergic use and the current steps towards their integration are illustrated. A case-study accounting for representative aspects of the Connect context is then elaborated in Section 4, which allows to provide preliminary illustrative examples of the analysis performed both off-line and on-line. Related work is briefly overviewed in Section 5, while conclusions and future perspectives are drawn in Section 6.

2 Background

In this section we provide some background material about dependability, performance and monitoring.

2.1 Dependability, Performance and Related Assessment Metrics

Dependability has been defined in the 90*'s* as the ability of a system to provide its intended services in a justifiable way [Lap95, ALRL04]. Such ability of the system is generally measured against the following attributes (see Figure 1):

availability, *reliability*, *safety*, *integrity*, *maintainability*. *Availability* is defined as the readiness for correct service and is generally computed as the ratio between the up-time of the system to the duration of the considered time period. *Reliability* is defined as the continuity of correct service and is typically expressed by using the notions of mean time between failures (MTBF) and mean time to recover (MTTR) for repairable systems, and with mean time to failure (MTTF) for non-repairable systems. *Safety* is the absence of catastrophic consequences. This attribute is a special case of reliability: a safe state, in this case, can be either a state in which a proper service is provided, or a state where an improper service is provided due to non-catastrophic failures. *Integrity* is defined as the absence of improper system state alterations. *Maintainability* is the ability to undergo modifications and repairs.

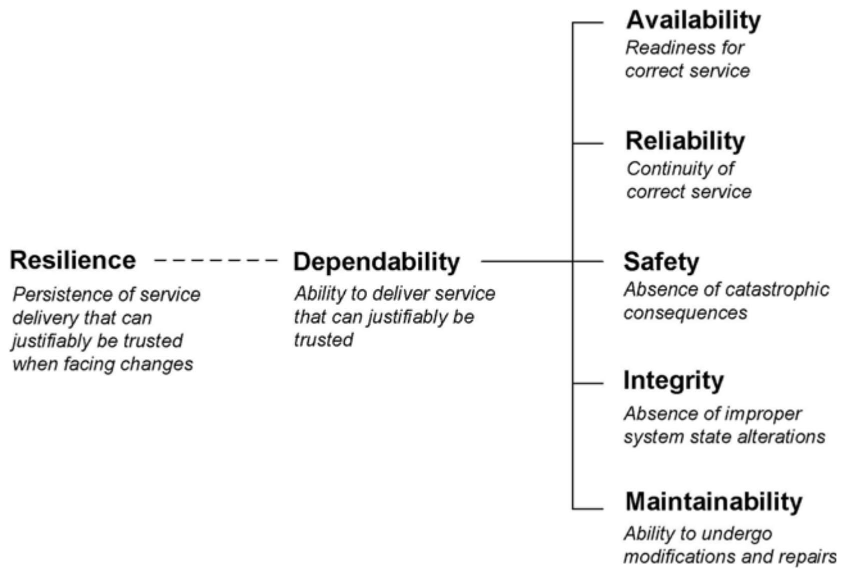

Resilience
Persistence of service delivery that can justifiably be trusted when facing changes
Dependability
Ability to deliver service that can justifiably be trusted
Availability
Readiness for correct service
Reliability
Continuity of correct service
Safety
Absence of catastrophic consequences
Integrity
Absence of improper system state alterations
Maintainability
Ability to undergo modifications and repairs

Fig. 1. Classical dependability attributes and resilience

Dynamic and evolvable systems generally need to cope with unanticipated conditions that might cause system failures. In these cases, the concept of dependability can be naturally extended to *Resilience*, i.e., the persistence of service delivery that can justifiably be trusted when facing changes [Lap08]. Possible changes can be classified according to their *nature* (e.g., functional, environmental, technological), *prospect* (e.g., foreseen, foreseeable, unforeseen), and *timing* (e.g., short, medium or long term).

Performance is the ability of a system to accomplish its intended services within given non-functional constraints (e.g., time, memory) [iee90]. Typically, performance of a system can be characterised with the following attributes (see

Figure 2): *timeliness*, *precision*, *accuracy*, *capacity* and *throughput*. *Timeliness* is the ability of the system to provide a service according to given time requirements, e.g., at a given time and within a certain time frame. *Precision* is the ability of the system to provide the same results when repeating measurements under unchanged conditions. *Accuracy* is the ability of the system to provide exact results, i.e., results that match the actual value of the quantity being measured. *Capacity* is the ability of the system to hold a certain amount of data or handle a certain amount of operations. *Throughput* is the ability to handle a certain amount of operations or data in a given time period.

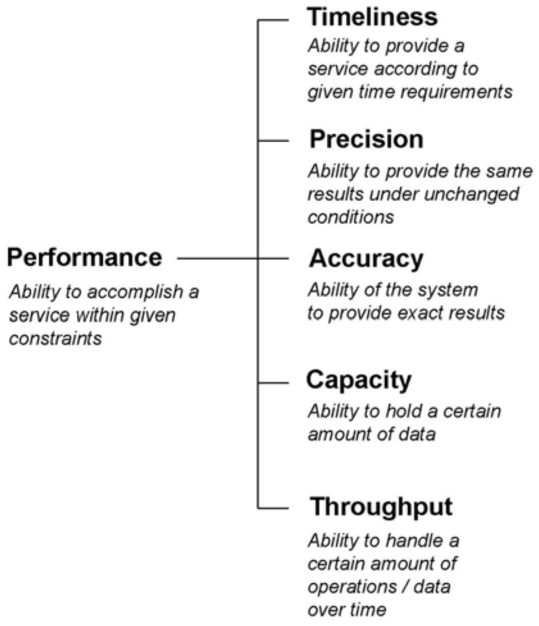

Performance
Ability to accomplish a service within given constraints
Timeliness
Ability to provide a service according to given time requirements
Precision
Ability to provide the same results under unchanged conditions
Accuracy
Ability of the system to provide exact results
Capacity
Ability to hold a certain amount of data
Throughput
Ability to handle a certain amount of operations / data over time

Fig. 2. Performance attributes

Quantification of dependability and performance attributes is of paramount importance in the process of determining whether a system meets its specification and to compare possible alternative design solutions leading to the most effective system realization. This is accomplished through the definition of appropriate metrics for the dependability and performance attributes. In general, a number of metrics can be defined for a given attribute; as an example, the following metrics allow to quantify *Availability*, that is the alternation between deliveries of proper and improper service:

- A(t) is 1 if service is proper at time t, 0 otherwise;
- E[A(t)] (Expected value of A(t)) is the probability that service is proper at time t;

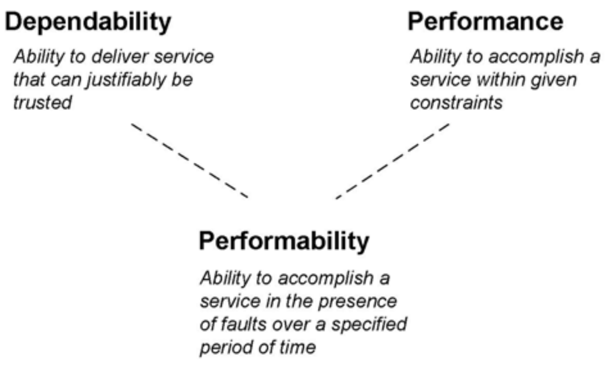

Dependability
Ability to deliver service that can justifiably be trusted
Performance
Ability to accomplish a service within given constraints
Performability
Ability to accomplish a service in the presence of faults over a specified period of time

Fig. 3. Performability and its relation with Dependability and Performance

- A(0,t) is the fraction of time the system delivers proper service during [0,t];
- E[A(0,t)] is the expected fraction of time the system delivers proper service during [0,t].

Similarly, *Performance* metrics typically include:

- the number of jobs processed per time unit, as a measure of throughput;
- the time to process a specific job, as a measure of the response time;
- the maximum number of jobs that may be processed per time unit, as a measure of the capacity.

Most practical *Performance* measures are very application specific, and measure times to perform particular functions or, more generally, the probability distribution function of the time to perform a function.

A measure of special interest introduced to evaluate degradable systems, i.e., systems that are still able to provide a proper service when facing faults, but with degraded level of performance, is *Performability.* This indicator combines the concepts of performance and dependability and represents the ability of a system to accomplish its intended services in the presence of faults over a specified period of time [Mey92]. Performability allows to evaluate different application requirements and to assess dependability-related attributes in terms of risk versus benefit.

2.2 Stochastic Model-Based Approaches for Early Prediction of Dependability and Performance Metrics

Fault forecasting and evaluation approaches are very suited to detect errors and deficiencies at design time, that could otherwise be very costly or even catastrophic when discovered at later stages.

Modelling is composed of two phases:

- The construction of a model of the system from the elementary stochastic processes that model the behaviour of the components of the system and

their interactions. These elementary stochastic processes relate to failures, to repair, service restoration and possibly to system duty cycle or phases of activity;
- Processing the model to obtain the expressions and the values of the dependability measures of the system.

Research in dependability analysis has developed a variety of models, each one focusing on particular levels of abstraction and/or system characteristics. As reported in [NST04], important classes of model representation include: *i)* Combinatorial Methods (such as Reliability Block Diagrams); *ii)* Model Checking; and *iii)* State-Based Stochastic Methods. In the Connect project, approaches at both points ii) and iii) are employed; in this chapter the emphasis is on State-Based Stochastic Methods, which support the explicit modelling of complex relationships (e.g., concerning failure and repair processes), and their transition structure encodes important sequencing information; for discussing Stochastic Model-Checking and for a compared evaluation of their usefulness in the Connect environment instead we refer to [CON10]. Concerning Combinatorial Methods, they have not been considered in Connect, since they do not easily capture certain features relevant for the project's context such as stochastic dependence and imperfect fault coverage,

State-Based Stochastic Methods use state-space mathematical models expressed with probabilistic assumptions about time durations and transition behaviours. State-based stochastic models can be classified in Markovian and non-Markovian according to the underlying stochastic process [CGL94, Hav01, Tri02]. A wide range of dependability modelling problems fall in the domain of Markovian models, for example when only exponentially distributed times occur. Markov chains (DTMC and CTMC) [How71, MFT00, Hav01, Tri02], Stochastic Petri nets (SPN) [Mol82, CBC$^+$93, Bal01] and Generalized Stochastic Petri nets (GSPN) [ABC84] are among the major Markovian models. However, there is also a great number of real circumstances in which the Markov property is not valid, for example when deterministic times occur; non-Markovian models are used for this type of problems. In past years, several classes of non-Markovian approaches have been defined [BT98], such as Semi-Markov Stochastic Petri Net (SMSPN's) [CGL94], Markov Regenerative Stochastic Petri Nets (MRSPN's) [CKT94] and Deterministic and Stochastic Petri Nets (DSPN's) [AC87]. Some major methods for analytically solving the non-Markovian models are discussed in [BPTT98, MFT00, Ger01]. A short survey on State-Based Stochastic Methods and automated supporting tools for the assisted construction and solution of dependability models can be found in [BCG05].

The proposal of an automated dependability analysis framework for dynamically ConnectEd systems will be discussed in 3.1. Two implementation of the analysis engine are being pursued: one based on the Stochastic Activity Networks (SAN) formalism and related Möbius tool, and the other on the PRISM tool. In this chapter, we focus on the Möbius implementation only; details on the PRISM-based implementation can be found in [CON11b]. The Stochastic Activity Networks (SAN) formalism is one of the most powerful (in term of modelling capabilities) stochastic extensions to Petri nets and is supported by the Möbius tool. SAN formalism and the Möbius tool are very commonly used in dependability analysis and therefore they have been initially chosen for dependability analysis in Connect. In the following, we provide some background on SAN and Möbius to get the reader more familiar with (part of the) dependability models we are going to define in this chapter.

Stochastic Activity Networks (SAN). SAN are stochastic extensions of Petri Nets introduced in [MM84] and formally defined in [SM02]. They have a graphical representation and consist of four primitive objects: *places*, *activities*, *input gates* and *output gates*. Places in SANs have the same interpretation as in Petri Nets, i.e., they hold tokens. The number of tokens in a place is referred to as the marking of that place, and the marking of the SAN is the set of all place markings.

There are two types of activities: instantaneous and timed. Timed activities represent actions that have a duration that impacts the performance of the modelled system, e.g., message transmission time, recovery time, time to fail. The duration of each timed activity is expressed via a time distribution function. Both instantaneous and timed activities may have *case probabilities*. Each case probability stands for a possible outcome of the activity, and can be used to model probabilistic aspects of the system, e.g., probability for a component to fail.

Gates connect activities and places. Input gates (indicated as red/grey triangles) are connected to one or more places and one single activity. They have a predicate, a boolean function of the markings of the connected places, and an output function. When the predicate is true, the gate holds. Output gates (indicated as black triangles) are connected to one or more places, and to the output side of an activity. If the activity has more than one case, output gates are connected to a single case. Output gates have only an output function. Gate functions (both for input and output gates) provide flexibility in defining how the markings of connected places change when the delay represented by an activity expires.

Properties of interest. Properties of interest are specified with *reward functions*. Each reward function is a C++ function that specifies how to measure a property on the basis of the marking of the SAN. There are two kinds of reward functions: *rate reward* and *impulse reward*. Rate rewards can be evaluated at any time instant. Impulse rewards are associated with specific activities and they can be evaluated only when the associated activity completes. Measurements can be conducted at specific time instants, over periods of time, or when the system reaches the steady state.

Möbius. Möbius [CCD+01] provides an infrastructure to support multiple interacting modelling formalisms and solvers. The main features of the tool include:

- *Multiple modelling languages*, based on either graphical or textual representations. Supported model types include stochastic extensions to Petri nets (e.g. SAN), Markov chains and extensions, and stochastic process algebras.

- *Hierarchical modelling paradigm.* Models are built from the ground up. First the behaviour of individual components is specified, and then a model of the complete system is created by combining these components.
- *Customized measures of system properties*, with ability to construct detailed expressions that measure the exact information desired about the system (e.g., reliability, availability, performance, and security). Measurements can be conducted at specific time points, over periods of time, or when the system reaches steady state.
- *Study the behaviour of the system under a variety of operating conditions.* The functionality of the system can be defined as model input parameters, and then the behaviour of the system can be automatically studied across wide ranges of input parameter values.
- *Distributed discrete-event simulation.* The tool evaluates the custom measures using efficient simulation algorithms to repeatedly execute the system.
- *Numerical solution techniques.* Exact solutions can be calculated for many classes of models, and advances in state-space computation and generation techniques make it possible to solve models with tens of millions of states.

Möbius allows to combine (atomic) models to form the *Composed model.* To this purpose, it supports the two operators *Rep* and *Join* to compose sub-networks. Join is used to compose two or more SANs. Rep is a special case of Join, and is used to construct a model consisting of a number of replicas of a SAN. Models in a composed system interact via *Place Sharing.* Place Sharing is a composition formalism based on the notion of sharing places via an equivalence relation. It supports the transient and steady-state analysis of Markovian models, the steady-state analysis of non-Markovian DSPN-like models [Sha93], and transient and steady-state simulation. More information can be found in the web site: http://www.crhc.uiuc.edu/PERFORM.

2.3 Run-Time Analysis via Monitoring

The ultimate goal of Connect, i.e., achieving automated and eternal interoperability among heterogeneous and evolvable Networked System, strongly relies on the adoption of on-line approaches, and therefore on a pervasive monitoring infrastructure.

More in general, as systems become more and more dynamic, distributed and evolvable, the capability of effectively gathering run-time information about their execution and/or their surrounding environment becomes an indispensable tool for many functionalities. Schroeder [Sch95], for example, identified the following seven monitor functionalities: Control, Correctness checking, Debugging&Testing, Dependability, Performance evaluation, Performance enhancement, and Security. In modern applications certainly further functionalities can be identified, such as Learning, Accounting, Trust management, and so on.

Although in this chapter we focus on the use of monitoring for dependability and performance evaluation, the monitoring infrastructure that we built in Connect (described in the next section) has been conceived to be general and flexible, and not restricted to this purpose. Here below, as a background we provide a basic introductory overview of monitoring concepts.

For the purpose of monitoring, the actions performed by the object under observation are abstracted into *events*. In particular, simple or primitive events are directly produced by the observed object, whereas *complex* events can be defined from simple events by using operators of a suitable *event algebra* [Zim99]. Event specification requires a careful design and configuration activity that is central to the overall setup of a monitoring system. In [SK88], events that may happen in a distributed system are divided in *local, non-local, global.* Local events are produced on a single node, which means that observing them does not require addressing the problems that are related to distribution and inter-node synchronization. Non-local events, on the other hand, are (composite) events whose observation requires considering and correlating events originated from more than a single node. Global events are a special case of non-local events, and require considering *all* the nodes of a system. Recognizing complex events in distributed loosely-coupled environments is not trivial [Fid96], as it requires establishing in which order two or more constituent events (originated from different nodes) happened. As noted by Lamport in his well-known 1978 paper, *in a distributed system it is sometimes impossible to say that one of two events happened first* [Lam78]. Fortunately, the observability problem does not necessarily arise in all distributed systems. For example, if one aims at identifying the service that takes the maximum average response time among a set of services, the problem is not really distributed, as the observation is in fact local and there is no need for aggregated interpretation.

Even though more than 20 years old, Joyce definition of monitoring as *the process of dynamic collection, interpretation, and presentation of information concerning objects or software processes under scrutiny* [JLSU87] remains still relevant and suggests some reflections.

Firstly, it qualifies monitoring as a *dynamic* activity, to underline that it is inherently concerned with the *execution phase* of a system, as opposed to (static) activities that are carried out in the development/coding phase. With this same meaning we also speak of *run-time* monitoring. Secondly, the definition identifies several different activities, namely collection, interpretation, and presentation, as part of monitoring. Each of these activities is meant to address a specific problem and may use dedicated techniques. This is why research on monitoring appears fragmented: the many works on monitoring are hard to compare as most of them focus only on some of the aspects of monitoring. A first attempt to overview the most important problems and issues about monitoring in distributed systems is [MSS94].

Monitor generic architecture. Figure 4 depicts the main architectural elements of a generic monitoring system. Elaborating on [MSS94], we identified the following five core functions.

Raw data collection. The lowest layer of a monitoring framework is realized by a set of sensors or probes: these fire a primitive event whenever the observed entity

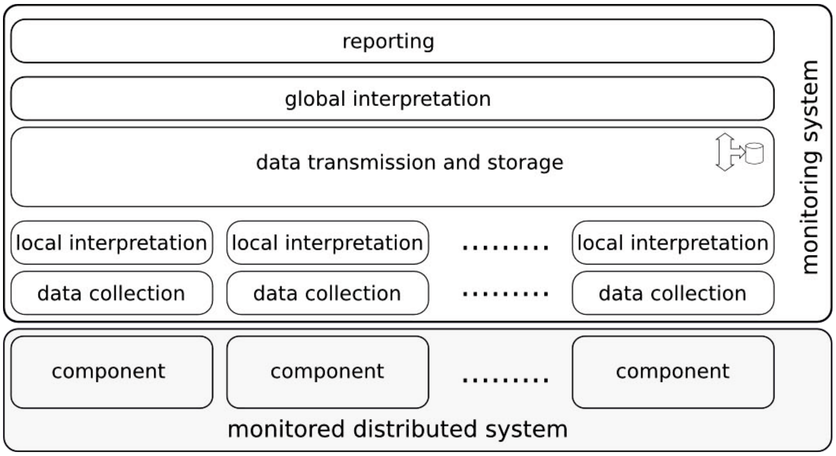


Fig. 4. Architectural elements of a monitoring system

performs some specified computation steps (actions). This is done locally on the entity under observation. Therefore, data collection is the function through which a monitor can produce the largest impact on the system under observation. We expand more on it later.

Local interpretation. The process of local interpretation is concerned with making sense (locally) of the information extracted by probes. This is achieved by applying a filter that extracts interesting sequences of events out of the raw data collected by probes. In practical implementations, data collection and filtering can overlap to some extent: a sort of rough preliminary filtering is done if the events emitted in the data collection step are not just the result of reaching a given point in the execution but also of some other logic or processing embedded in the probe.

Data transmission and storage. In distributed systems, the relevant events that are revealed locally need to be collected at one or more sites where they can be aggregated with analogous data coming from other nodes. Transmission may occur immediately, to reduce detection latency, or may be delayed, using buffering, e.g., to cope with network congestion (at the expense of memory occupation or CPU cycles if compression is used).

Global interpretation (also known as "correlation" or "aggregation"). This function makes sense of pieces of information that come from several nodes and puts them together in order to identify interesting conditions/events that can be observed only at an aggregated level. Architectures with more than one level of aggregation are possible, and are usually adopted when enhanced scalability is necessary [MCC04]. When moving from local to global observation, observability issues must be taken into account. Suitable timestamping and synchronization facilities must be used as appropriate.

Reporting. The information provided as the output of monitoring can be used for a variety of purposes and should be presented in a way that is meaningful to the "consumer" of the monitoring system. The consumer can be a piece of software itself or a human. In both cases the results of the final interpretation phase must undergo an elaboration in order to express output data in a suitable format. In the former case, this format should be machine-readable; in the latter, it must be shown either as a textual report or it may use interactive GUIs, graphics, animations and so on.

The first core function, data collection, determines the monitoring system *intrusiveness* [Sch95], i.e., the level of interference imposed upon the observed application. Intrusive monitors may alter the behaviour that they want to observe. This phenomenon is referred to as the probe effect (or sometime the "Heisenberg effect") [Gai86]. It concerns especially monitoring of performance-related characteristics, but may also impact functional properties, since the process might alter the timing of events and therefore cause wrong behaviour that otherwise would not happen [JLSU87]. Analogously, faults (that would have happened otherwise) can be masked as an effect of the interplay of the subject system and the monitoring.

The collection of data can be done according to two styles [HBPU06]:

- *Instrumentation:* some code is inserted in the application to be monitored in order to emit an event when the control flow reaches a certain point in the execution. This can be realized in different forms at different levels of abstraction. For example, source code instrumentation techniques include statements in the original program to be monitored. Their outcome can be guarded by a condition, whereby it is possible to refine the event definition and to emit an event only when the guard condition is satisfied.
- *Interception:* when adopting this style, data collection is achieved through a proxy-like probe that is put on the wire and snoops interactions, as in [BGG04]. Although this approach may not be as flexible as the previous, it has the advantage of being non-intrusive, therefore it is well suited in other contexts where the control over the system is distributed/partioned across several organizations.

Orthogonally to the instrument/intercept criterion, the techniques for collecting monitoring data can also be distinguished according to whether the collection is based on sampling or complete executions are observed. Most sampling approaches sample on a time basis; however certain (composite) events may go undetected if some of the constituent events are discarded by the sampling. A possible way to address this problem is by sampling in space, rather than in time, i.e., by alternating the processes (or components) that are chosen as the target of monitoring, in such a way that, locally to the target, the observation is complete and no event is discarded.

3 Dependability Assessment Approach in CONNECT

Assessment techniques are sought in CONNECT to ensure that Networked Systems as well as the generated bridging CONNECTors satisfy specified levels of accomplishment for dependability and performance requirements, according to pertinent metrics. In the following, we present two approaches under development in CONNECT for this purpose: the Dependability&Performance Enabler (DEPER) and the Monitoring Enabler (GLIMPSE). The two approaches are first described individually and then their combined use to enhance dependability and performance assessment of the system under analysis is discussed.

3.1 DePer

DEPER provides support to the definition of a CONNECTor that allows NSs to interact with a desired level of dependability and performance properties.

Before presenting the architecture of this Enabler, we briefly discuss its relations with other Enablers of the CONNECT architecture. In [GGB+11], a complete overview of CONNECT Enablers and their role is provided. Adopting a DEPER-centric view, here we restrict to those having input-output relations with DEPER, as shown in Figure 5 (to make the chapter self-complete, the role of relevant Enablers has also been synthetically recalled in Section 1).

According to the CONNECT vision, a Networked System broadcasts a *connect request* whenever a new connection to a service is needed. The connect request contains a description of the requested service together with a specification of the required dependability and performance level for the service. When Discovery detects a connect request, it looks for available Networked Systems that can provide the requested service. If such systems are found and operate a communication protocol different from that of the Networked System that made the connect request, Discovery triggers the process of creating a suitable CONNECTor

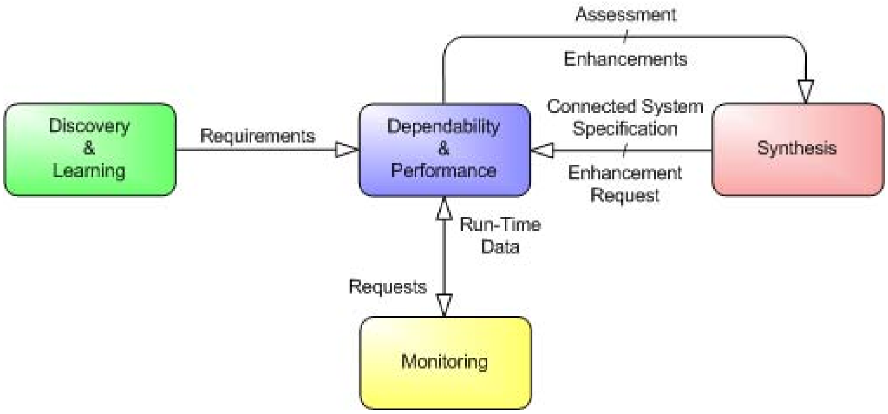

Assessment / Enhancements
Discovery & Learning
Requirements
Dependability & Performance
Connected System Specification / Enhancement Request
Synthesis
Run-Time Data
Requests
Monitoring

Fig. 5. Input-Output Relations between DEPER and the Other Enablers

that enables interoperation. The Synthesis Enabler, on the basis of the specification of the communication protocols, produces a mediating CONNECTor. Before CONNECTor deployment, Synthesis activates DEPER to evaluate if the CONNECTed system that will be obtained satisfies the non-functional requirements expressed by the Networked Systems. If the non-functional requirements are satisfied, the CONNECTor is deployed; otherwise, Synthesis is supported by DEPER in the definition of possible enhancements that can be applied. To take into account dynamic system changes once the CONNECTor is deployed, the Monitoring Enabler, if requested by other Enablers, continuously observes the run-time behaviour of the CONNECTed system and provides them related information, in accordance with received requests.

Therefore, as shown in Figure 5, the joint activity of Discovery and Learning provides the dependability requirements; Synthesis provides the specification of the CONNECTed system, and possibly requests a dependability enhancement; Monitoring provides run-time data on the execution of the deployed CONNECTor. The dependability and performance assessment and the enhancements produced by DEPER are used by Synthesis.

The architecture of the DEPER Enabler is shown in Figure 6 and also preliminarily described in [MMDGar]. Currently, this Enabler accommodates dependability and performance analysis performed through both the stochastic state-based and the stochastic model-checking approaches. Actually, the architecture is general and other analysis methods could be easily included by specifying and implementing an appropriate Analysis Engine module. The Selector and Aggregator modules, at the entrance and exit of the architecture, allow the selection of the analysis method and the aggregation of the analyses results (in case more than one method is applied), respectively. More details on each module are provided in the following.

At the time of writing, a prototype which partially implements the DEPER architecture is under development (http://dcl.isti.cnr.it/DEA/). It is based on the SAN formalism and the Möbius tool, already introduced in Section 2. For those modules already considered in the implementation, some details are also provided in the following description.

3.2 Selector

The Selector module activates, depending on the characteristics of the specification of the CONNECTed system and of the requirements, the most suitable analysis engine among those available to the Enabler. In fact, the employed engines implement different approaches to analyse dependability and performance properties of a CONNECTed system. Each approach has its own advantages regarding modelling capability, specification of properties, and scalability; hence, besides using the different engines to cross validate the results and to improve the confidence in the correctness of the models, they actually complement each other. In the study conducted during the first year [CON10], the characteristics of the stochastic model checking and state-based stochastic methods evaluation approaches have been already pointed out.

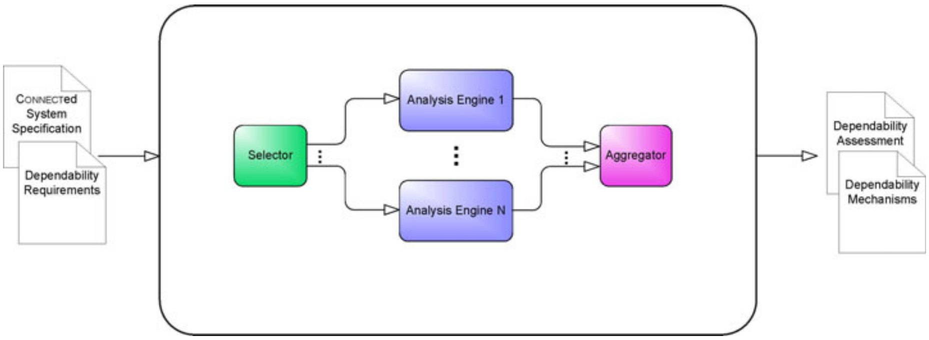


Fig. 6. Architecture of the Dependability&Performance Analysis Enabler

3.3 Aggregator

The Aggregator module is in charge of selecting the analysis results to be provided in output to the Synthesis Enabler, in case more than one Analysis Engines have been activated for a CONNECTed system specification. Therefore, when only one kind of analysis is performed, based on the choice made by the Selector module, Aggregator just conveys the analysis results to the output interface of DEPER. Instead, when more analysis methods are activated, their results are collected by Aggregator and elaborated according to some criteria to derive the output results.

The first step to be performed is a comparison among the values provided by the different methods to check whether they are in agreement (within a certain tolerance degree, to cope with natural dissimilarities inherent to the use of different methods). In the case of a matching comparison, the reliance in each of the employed approaches is increased (cross-validation) and all the analysis results can be equally considered valid, so anyone of them can be output as final analysis values. Alternatively, some form of mediation could be made on the obtained multiple results (e.g., the average), to balance the effects of single method's approximations. A mismatch, instead, would be the symptom of erroneous/-too inaccurate analysis by at least one of the applied methods. Let us recall that DEPER in CONNECT is based on an automated procedure, starting from given specifications of the CONNECTed system and of the dependability and performance requirements, partially implemented at the current stage. Therefore, once fully automated, we expect that the case of mismatch would be removed by construction; however more investigations are necessary on this issue. The implementation of this module has been deferred, at the moment.

3.4 Dependability&Performance Analysis Engine

The Dependability&Performance Analysis Engine is logically split into five main functional modules (see Figure 7): Builder, Analyser, Evaluator, Enhancer and Updater.

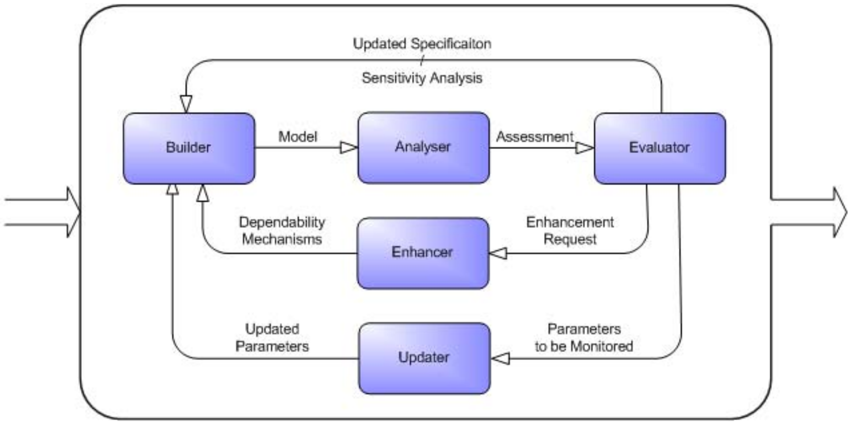


Fig. 7. Architecture of the Dependability&Performance Analysis Engine

Builder. The Builder module takes in input the specification of the CONNECTed system from Synthesis, and the dependability and performance requirements from Discovery/Learning. The module produces in output a dependability and performance model of the CONNECTed system suitable to assess the given dependability and performance requirements.

Specification of the CONNECT*ed system.* With reference to recent works on synthesis of mediating CONNECTors [SI10] and automata discovery/learning [RSB05], the specification of the CONNECTed system is given with Labelled Transition Systems (LTSs) annotated with non-functional information necessary to build the dependability and performance model of the CONNECTed system. An LTS is an abstract machine that represents the sequence of actions performed by the system. Formally, an LTS is a tuple $(\mathcal{S}, \mathcal{S}_0, \mathcal{L}, \mathcal{T})$, where $\mathcal{S}$ is a set of states, $\mathcal{S}_0 \subseteq \mathcal{S}$ is a set of initial states, $\mathcal{L}$ is a set of labels, and $\mathcal{T} \subseteq \mathcal{S} \times \mathcal{L} \times \mathcal{S}$ is a transition relation. Annotations include, for each labelled transition, the following fields: *time to complete*, *firing probability*, and *failure probability*. The values for these parameters could be exact values or ranges of values (ranges are especially appropriate when the exact estimate is not possible, given uncertainties of the environment).

Dependability and performance requirements. In our architecture, the dependability and performance properties required by the Networked Systems are translated by Discovery/Learning into *metrics* and *guarantees*. Metrics are arithmetic expressions that describe how to obtain a quantitative assessment of the properties of interest of the CONNECTed system. They are expressed in terms of transitions and states of the LTS specification of the Networked Systems. Guarantees are boolean expressions that are required to be satisfied on the metrics.

Dependability and performance model. The dependability and performance model of the CONNECTed system is specified with a formalism that allows to describe complex systems that have probabilistic behaviour, e.g., stochastic processes.

Implementation. The prototype implementation of the Builder module takes in input the LTS of the connected system described with Finite State Processes (FSP) [MK06]. The dependability model of the system is specified with Stochastic Activity Networks (SANs), already introduced in section 2. The SAN model is obtained from the LTS model by using the theory of regions [ER90]. A region identifies a set of states in the LTS such that all transitions with the same label either enter, exit, or never cross the boundary of the region. Each region in the LTS corresponds to a place in the derived SAN model, and each labelled transition in the LTS corresponds to an activity in the SAN model. A similar approach has already been used in other works to translate LTSs into Petri Nets (see, for instance, [CKLY98], [BS02] and [CCK09]). In order to have a well-defined probabilistic model, non-deterministic choices among k transitions outgoing from an LTS state are mapped in the SAN model into instantaneous activities with k case probabilities. The metric is an arithmetic expression that may contain a predefined set of functions (see Table 1 for some examples). The guarantee is given by a boolean expression on the metric and a set of constraints on the connected system model (e.g., constraints on the time frame of evaluation of the metric). Statistical operators (e.g., *mean* and *variance*), comparison and logical operators can be used in the expression.

Table 1. Examples of predefined functions that can be used in the metric expression

Function	Description
$timeFrame(s) : \mathcal{S} \to \mathbb{R}^+$	returns the interval of time when the system is in state s
$minTimeStamp(tr) : \mathcal{T} \to \mathbb{R}^+$	returns the first instant of time when transition tr fires
$avgTimeStamp(tr) : \mathcal{T} \to \mathbb{R}^+$	returns the average instant of time when transition tr fires
$maxTimeStamp(tr) : \mathcal{T} \to \mathbb{R}^+$	returns the last instant of time when transition tr fires
$\#(tr, t1, t2) : \mathcal{T} \times \mathbb{R}^+ \times \mathbb{R}^+ \to \mathbb{N}$	returns the number of times transition tr fires during the time frame $[t1, t2]$
$\#(l, t1, t2) : \mathcal{L} \times \mathbb{R}^+ \times \mathbb{R}^+ \to \mathbb{N}$	returns the number of times transitions with label l fire during the time frame $[t1, t2]$

Analyser. The Analyser module takes in input the dependability and performance model from the Builder module and the dependability and performance requirements from Discovery/Learning. The module builds a reward model, i.e., a model that enables a quantitative assessment of the metrics of interest, and makes use of a solver engine to obtain a quantitative assessment of the dependability and performance metrics.

Reward model. The reward model is the dependability and performance model extended with reward functions. Reward functions allow to specify properties of interest: they return a value depending on the system state, and can be evaluated either at an instant of time or accumulated over a time frame.

Solver. The solver engine evaluates the reward functions defined in the reward model. The evaluation can be performed either through analytical approaches or through simulation, depending on the metrics under evaluation and on the mathematical representation of the involved phenomena.

Implementation. The prototype implementation of the Analyser is based on Möbius, already introduced in section 2. In Möbius, each reward function is a C++ function that returns a value depending on the marking of the SAN. There are two kinds of reward functions: *rate rewards* and *impulse rewards*. Rate rewards are used to implement time-based reward functions. Impulse rewards are used to implement action-based reward functions, i.e., they are associated with specific activities and can be evaluated only when the associated activity completes. The reward functions are automatically derived from the metrics expression as follows: the metric is mapped into its syntax tree to decompose the metric into a combination of basic functions; the basic functions are translated into C++ functions by using a predefined repository of function templates (currently under construction). For instance, with reference to the functions shown in Table 1, a rate reward template is used to translate $timeFrame(s)$, while an impulse reward template is used to translate $\#(tr, t1, t2)$. Then, the quantitative assessment of the metric is obtained from the assessment of the reward functions by merging the results according to the arithmetic operations specified in the syntax tree of the metric expression.

Evaluator. The Evaluator module reports to Synthesis if the CONNECTed system satisfies the dependability and performance requirements provided by Discovery/Learning. In the case of requirements mismatch, Evaluator sends a warning message to Synthesis, and may receive back a request to evaluate if enhancements can be applied to improve the dependability (and/or performance, depending on the received request) level of the CONNECTed system.

In view of the synergic cooperation with the monitoring infrastructure, this module also informs the Updater module, which is in relationship with the Monitor Enabler, about the model parameters for on-line observation.

Requirements mismatch. If the requirements are not satisfied, Evaluator may receive a request to explore one of the following three directions for improvements:

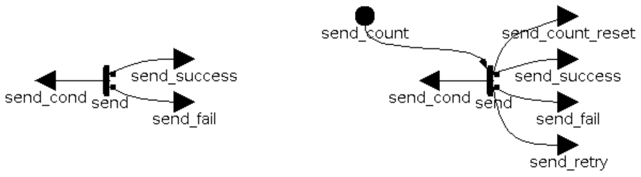
1. Update the specification of the CONNECTor to take into account an alternative CONNECTor deployment (e.g., a deployment that uses a communication channel with lower failure rate). Upon receiving this request, the Evaluator triggers a new analysis that considers the updated specification of the CONNECTor.

2. Enhance the specification of the Connector by including dependability mechanisms, which are counter-measures to contrast failure modes affecting performance and/or dependability metrics (e.g., a message retransmission technique). Upon receiving this request, it is first necessary to understand which are the failure probabilities mostly impacting on the metrics evaluation, so as to include primarily dependability mechanisms capable of limiting the effects of such highly impacting failures. To this end, Evaluator builds a sensitivity analysis campaign to instruct the Builder module on the creation of dependability and performance model variants, each of which considers a specific subset of failure probabilities, among those foreseen. Whenever a variant is generated, the Analyser module performs the assessment of the metrics on the generated model. Evaluator collects the analysis results and, after all variants have been analysed, produces a ranking of the failure probabilities. This ranking is used by Evaluator to iteratively activate the Enhancer module until one of the following conditions is met: the guarantees are satisfied, or Enhancer signals that all possible dependability mechanisms have been explored.
3. Apply a combination of the previously mentioned enhancements.

Enhancer. The Enhancer module is activated by Evaluator when the guarantees are not satisfied and Synthesis makes a request to enhance the Connector with dependability mechanisms. Enhancer is instructed by the Evaluator module on the requirements mismatch and the failure probability that needs to be improved. Then, it performs the following actions: (i) selects the dependability mechanisms that can be employed, among those available, to improve the failure probability indicated by the Evaluator module; (ii) instructs the Builder module on the application of the selected dependability mechanisms in the Connected system model (one model variant will be generated for each dependability mechanism selected, generally only one) and triggers a new analysis for each of the generated models.

Dependability mechanisms. Typically, dependability mechanisms are based on the application of redundancy, e.g., duplication of system channels, or retry of message transmissions over system channels. The dependability mechanism, in this context, will be embedded in the synthesised Connector, because Networked Systems are not under the control of the framework. Nevertheless, the dependability mechanisms embedded in the Connector can be employed to improve, to some extent, the dependability and performance level of the Networked Systems. For example, the reliability level of a transmission performed by a Networked System can be improved through timeouts or message retransmissions applied at the Connector level.

Implementation. We developed ad hoc dependability models for a set of relevant dependability mechanisms, and a set of rules to automate the application of the mechanisms in the SAN model of the connected system. The ad hoc models can be parametric; for instance, a retransmission mechanism is parametric with



(a) Basic model (b) Enhanced model

Fig. 8. Example of dependability mechanism and application rule

respect to the maximum number of allowed retransmissions. As an example of mechanism and application rule, in Figure 8 we graphically show a retransmission mechanism and how to modify the original model in order to apply message retransmissions to a send operation. Specifically, the original model contains a timed activity `send` that models the send operation, an input gate `send_cond` that specifies the enabling condition of the activity, and two output gates, `send_success` and `send_fail`, that specify the output functions in the case of correct and faulty behaviour. The enhanced model is obtained from the original model by adding the following elements: a place `send_count`, with initial marking the maximum allowed number of retransmissions; an output gate `send_count_reset`, which resets the marking of `send_count` to its initial value when the `send` succeeds; an output gate `send_retry`, which reactivates `send` as long as `send_count` contains tokens, and resets the marking of `send_count` after performing all retransmission attempts.

In a more structured vision, the dependability mechanism(s) suitable to improve on a given failure probability are determined through an ontology of dependability mechanisms, such as that reported in [ReS08]. The definition of such an ontology is planned as future work.

Updater. The Updater module interacts with the Monitoring Enabler to refine the accuracy of model parameters through on-line observations. Inaccuracy of the non-functional values used in the off-line analysis at CONNECTor design time is mainly due to two possible causes: i) limited knowledge of the NSs characteristics acquired by Learning/Discovery Enablers; ii) evolution along time of the NSs, as naturally accounted for in the CONNECT context.

Updater receives inputs from both internally to DePer (from the Evaluator) and externally (from the Monitor Enabler). From the former, for each CONNECTor ready to be deployed it receives the model parameters to convey to the Monitor for run-time observations. From the latter, it receives a continuous flow of data for the parameters under monitoring relative to the different executions of the CONNECTor. Accumulated data are processed through statistical inference techniques. If, for a given parameter, the statistical inference indicates a discrepancy between the on-line observed behaviour and the off-line estimated value used in the model resulting into a significant deviation of the performed analysis, a new analysis is triggered by instructing the Builder module to update the CONNECTed system model.

Details on the implementation between Updater and Monitoring are illustrated in Subsection 3.6.

Statistical Inference Techniques. As reported in [Tri02], methods of statistical inference applied to a collection of elements under investigation (called population), allow to estimate the characteristics of the entire population. In our case, the collection of values relative to each parameter under monitoring constitute a population to which such techniques are applied.

3.5 GLIMPSE

In Connect we developed a monitoring infrastructure, called Glimpse, aimed at covering the main function[MSS94] for the monitoring discussed in section 2.3. We tried to implement these five core function in a modular and flexible way, aiming to support behavioural learning, performance, reliability assessment, security and trust also in non-Connect context.

Glimpse is an acronym for "Generic fLexIble Monitoring based on a Publish-Subscribe infrastructurE". The architecture of Glimpse is shown in Figure 10; details of each component are in the following.

Probes. There are entrusted to *Collection* and *Local interpretation* functions. Probes are usually realized by injecting code into the existing software or by using a proxy. The probes that we used in Glimpse are already injected into the ConnECTor during its synthesis phase. When primitive events occur into the software, probes send them to the Monitoring Bus component (described below).

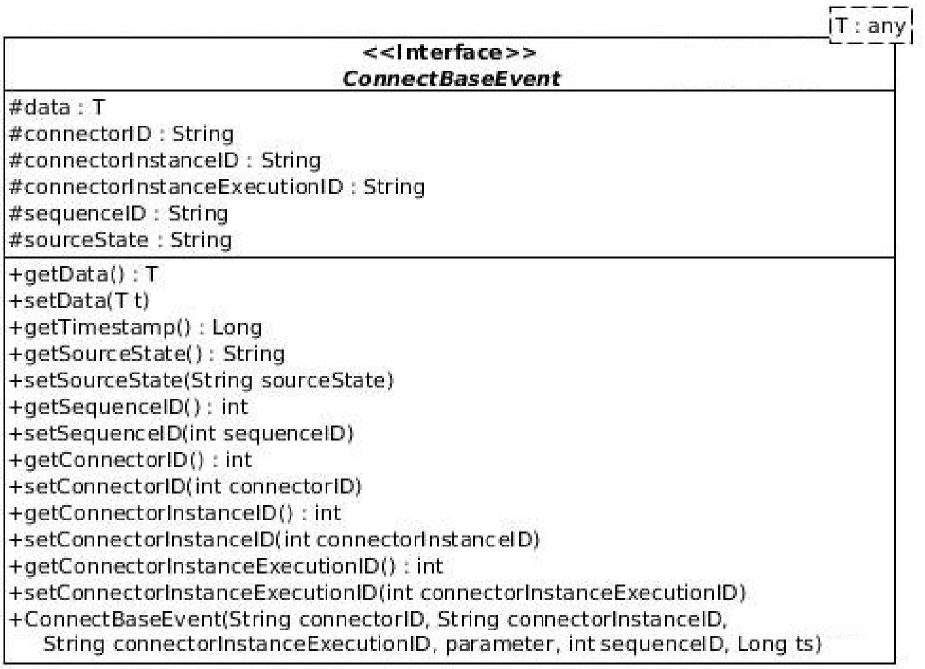
An event, in the current context, represent a transition between two states of an LTS. In our implementation we collect all the information that may be useful at Glimpse to infer complex event occurrences for the analysis in the `ConnectBaseEvent` interface.

The event description is shown in figure 9.

Examinating more in details some of the parameters composing the `Connect BaseEvent` interface we can find:

- `connectorID` : defines the identity of the ConnECTor
- `connectorInstanceID` : used to define the execution of the ConnECTor

Monitoring bus. The monitoring bus is the communication backbone where all information (events, requests, responses) is sent on by: Probes, Enablers, Complex Event Processor and by all the services using Glimpse. To obtain a better decoupling and to keep asynchronous communication, in our implementation we decided to adopt public-subscribe paradigm, which is implemented with messages queue through ServiceMix4 and Java Message Service. There are many commercial products that implement Enterprise Service Bus, we prefer ServiceMix for its strong compatibility and simplicity interacting with the rule engine used for this prototype.



```
T : any
<<Interface>>
ConnectBaseEvent
#data : T
#connectorID : String
#connectorInstanceID : String
#connectorInstanceExecutionID : String
#sequenceID : String
#sourceState : String
+getData() : T
+setData(T t)
+getTimestamp() : Long
+getSourceState() : String
+setSourceState(String sourceState)
+getSequenceID() : int
+setSequenceID(int sequenceID)
+getConnectorID() : int
+setConnectorID(int connectorID)
+getConnectorInstanceID() : int
+setConnectorInstanceID(int connectorInstanceID)
+getConnectorInstanceExecutionID() : int
+setConnectorInstanceExecutionID(int connectorInstanceExecutionID)
+ConnectBaseEvent(String connectorID, String connectorInstanceID,
   String connectorInstanceExecutionID, parameter, int sequenceID, Long ts)
```

Fig. 9. ConnectBaseEvent Interface

However, in designing the Glimpse architecture, we adopted a model-driven which allow Glimpse to use different rule languages. The handling of the Monitoring Bus is devoted to the Manager component, analysed below.

The usage of a messaging system in Enterprise Service Bus (ESB) allows Glimpse to use a variety of protocols such as HTTP/SOAP and REST. Moreover, with the usage of JBI [jbi] components, Glimpse interact even with legacy systems, binary transports, document-oriented transports, and Remote Procedure Call [rpc] systems. Adopting a messaging system reduces execution bottlenecks that might occur using Remote Procedure Call or Database-centric architecture.

Complex Event Processor. The Complex Event Processor (CEP) is the rule engine that allows to infer complex events from primitive sent on the Monitoring Bus by Probes.

The CEP is instructed at runtime by the Manager component that, after analysing the consumer (Enabler) request expressed sending a JMS message on wich payload is written in XML using the `ComplexEventRuleActionList` schema (see Listing 1), load the new inference rules on the engine. There are several rule engines that can be used for this task (like Drools Fusion [dro], RuleML [rul]), in the existing prototype wu used Drools Fusion.

The schema of `complexEventActionList` is shown in Listing 1.

The `complexEventActionList` specification supports the use of heterogeneous rule languages, as it is natively unbound to any. Examinating the Listing 1, at line 30, we found the RuleBody field that is used by the Enabler to set the Drools rule for the requested evaluation. Into the field RuleType (line 32), the Enabler will set the type of rule language requested for the evaluation, this will allow to use more rule languages even at run-time.

Consumer. In GLIMPSE, a `Consumer` may be a learning engine, a dependability analyser or a simple customer that requests some information to be monitored. The basic requirement to interact with GLIMPSE is to be able to send/receive JMS messages and to raise correct query to the inference engine (CEP). The `Consumer` sends a request to the Manager using the Monitoring Bus and waits for the evaluation results on a dedicated response channel provided and notified by the Manager.

Manager. It manages all the communications among its components. The Manager component is the orchestrator of all the GLIMPSE architecture. Specifically, the Manager fetches requests coming from Consumers (Enablers), analyses them and instructs the Probes. Then, it instructs the CEP Evaluator, creates and notifies to the Consumer a dedicated channel on which it will provide results produced by the CEP Evaluator. (For more information about interaction, sec:3.6).

The most valuable support provided by the `Manager` is the handling of all the knowledge base loaded into the CEP.

GLIMPSE *Implementation.* A prototype of GLIMPSE is available for public download at http://labse.isti.cnr.it/tools/glimpse.

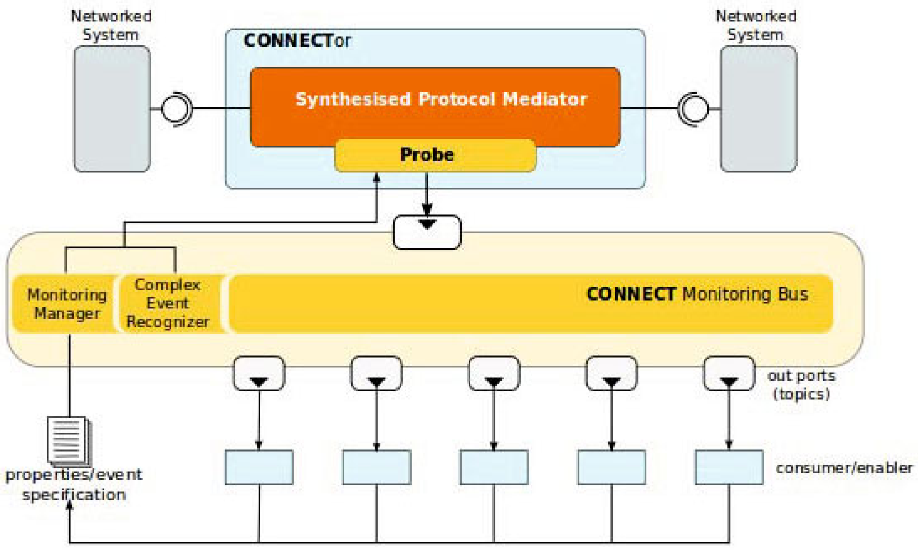


Fig. 10. The architecture of GLIMPSE

3.6 Integrated Run-Time Analysis

Synergic use of DePer and Glimpse is pursued to allow automated refinement of dependability and performance analysis through inspection of run-time data, as preliminarly described in [BCDG$^+$ar]. Precisely, feedbacks from run-time executions of the CONNECTed system as collected from Glimpse are used by DePer to enhance the accuracy of model parameters adopted in the analysis performed at design time.

The interactions between DePer and Glimpse Enablers start after the DePer Enabler determines that the synthesised CONNECTor satisfies the required dependability and performance level. Specifically, after the analysis phase, if compliance with requirements is verified with the consequent deployment of the CONNECTor, DePer informs the Glimpse on which are the parameters (among those used in the dependability analysis) relative to CONNECTor and NSs, that must be kept under observation at run-time. Glimpse, upon receiving the request, properly instructs the probes embedded in the CONNECTor.

Run-time data relative to parameters under observation are sent by the Glimpse to DePer. DePer, through its Updater module, continuously performs statistical analyses on the collection of data received to verify whether the accuracy of the model parameters used in the analysis is good enough for the analysis results to be still valid, or the CONNECTor no longer satisfies the requirements and needs adjustments. In this latter case, a new analysis adopting the updates parameters values is triggered.

Figure 11 shows the interaction between DePer and Glimpse. DePer is shown inside the dotted box at the top of the Figure, while Glimpse is shown again in a dotted box at the bottom left side of the Figure. For clarity, only relevant modules involved in the cycle with Monitoring, are shown. Inside DePer, the activities from receiving the CONNECTed System specifications till conclusion of the evaluation phase are represented. Since we want to show the interaction between DePer and Glimpse, we depicted the case where the dependability and performance requirements are met, so the Evaluator module reports to Synthesis that the CONNECTor can be deployed and triggers Updater on sending monitoring requests to Glimpse. Such requests are received by the Manager component of Glimpse through a service channel on the Monitoring Bus, which instructs the Probes and the Complex Event Processor and creates a dedicated communication channel on the Monitoring Bus, used to provide results to DePer.

Then, Probes start intercepting events of interest, when they occur. If DePer requests consist of complex events, their composing primitive events are elaborated by the rule engine Complex Event Processor, and resulting values are computed.

Responses to monitoring requests so determined are sent to DePer through a dedicated channel on the Monitoring Bus.

Once the Updater module of DePer receives the run-time data from Glimpse, it applies to them statistical inference techniques to determine the actual values of the corresponding model parameters. In case the newly determined values are

```xml
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema" targetNamespace="http://labse.isti.cnr.it/glimpse/xml/ComplexEventRule" xmlns:tns="http://labse.isti.cnr.it/glimpse/xml/ComplexEventRule" elementFormDefault="qualified">


    <element name="ComplexEventRuleActionList" type="tns:ComplexEventRuleActionType"></element>

    <complexType name="ComplexEventRuleActionType">
        <sequence>
            <element name="Insert" type="tns:ComplexEventRuleType"
                 maxOccurs="unbounded" minOccurs="0">
            </element>
            <element name="Delete" type="tns:ComplexEventRuleType"
                 maxOccurs="unbounded" minOccurs="0">
            </element>
            <element name="Start" type="tns:ComplexEventRuleType"
                 maxOccurs="unbounded" minOccurs="0">
            </element>
            <element name="Stop" type="tns:ComplexEventRuleType"
                 maxOccurs="unbounded" minOccurs="0">
            </element>
            <element name="Restart" type="tns:ComplexEventRuleType"
                 maxOccurs="unbounded" minOccurs="0">
            </element>
        </sequence>
    </complexType>

    <complexType name="ComplexEventRuleType">
        <sequence>
            <element name="RuleName" type="string" maxOccurs="1" minOccurs="1"></element>
            <element name="RuleBody" type="string" maxOccurs="1" minOccurs="0"></element>
        </sequence>
        <attribute name="RuleType" type="string"></attribute>
    </complexType>
</schema>
```

Listing 1. ComplexEventRuleActionList Schema

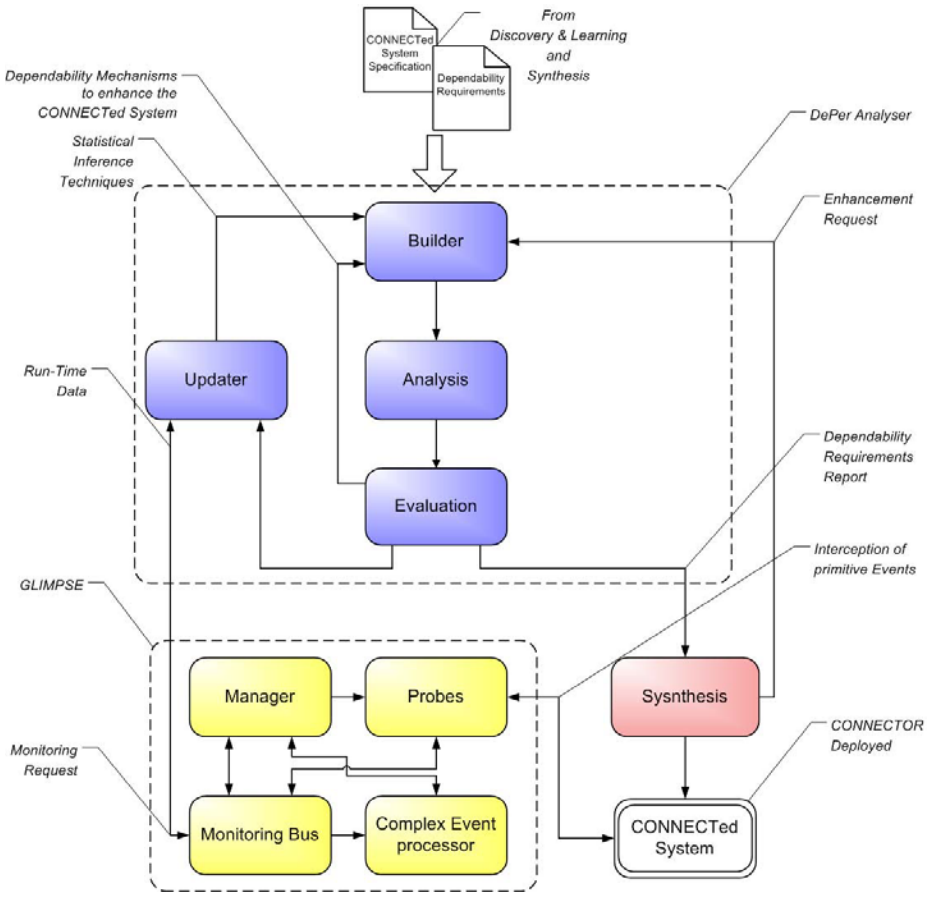

CONNECTed System Specification
Dependability Requirements
From Discovery & Learning and Synthesis
Dependability Mechanisms to enhance the CONNECTed System
Statistical Inference Techniques
DePer Analyser
Enhancement Request
Builder
Updater
Analysis
Evaluation
Run-Time Data
Dependability Requirements Report
Interception of primitive Events
GLIMPSE
Manager
Probes
Sysnthesis
CONNECTOR Deployed
Monitoring Request
Monitoring Bus
Complex Event processor
CONNECTed System

Fig. 11. Interactions between DePer and Glimpse

outside the range assumed in the analysis at design time, the analysis model is updated with the new values and solved again. If the new analysis evidences that the deployed Connector needs adjustments, a new synthesis-analysis cycle is started in cooperation with the Synthesis Enabler and a notification is sent to Glimpse about stopping monitoring the no more satisfactory Connector.

Implementation. In the implementation performed so far, DePer and Glimpse interact by using a Publish/Subscribe protocol. The interaction pattern is shown as a sequence diagram in Figure 12 where we intentionally left out system start-up operations (for a more detailed sequence diagram, see [CON11a]). Whenever Monitoring Enabler receives a request message on the service channel (see message 2 on Figure 12), a new channel dedicated to the requesting Enabler is set up to communicate the monitored values.

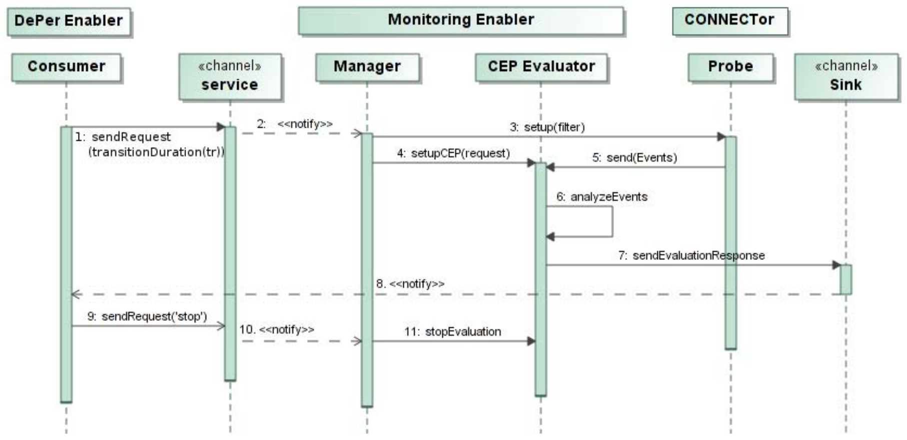


Fig. 12. Sequence Diagram of the Basic Interaction Pattern between DePer and Glimpse

Glimpse sends response messages to DePer Enabler as soon as the aspect of interest is available (see message 8 on Figure 12).

The two Enablers exchange JMS messages whose payload is expressed in XML language. The payload of the XML, contains a `ComplexEventRuleActionList` xml object, which determines a lists of possible actions to execute on the Monitoring Enabler knowledge base. The schema of ComplexEventRuleActionList is shown in Listing 1.

4 Example

In this section we first introduce an example scenario and then we show how we apply the presented approaches to it.

4.1 The Terrorist Alert Scenario

We consider the Connect Terrorist Alert scenario [CON11c], depicting the critical situation that during a show in the stadium, the control center spots one suspect terrorist moving around. The alarm is immediately sent to the Police.

Policemen are equipped with ad hoc handheld devices which are connected to the Police control center to receive command and documents. Precisely, the policemen can share documents with the Police control center and with other policemen through a *SecuredFileSharing* application, for example a picture of a suspect terrorist.

Unfortunately, the suspect is put on alert from the police movements and tries to escape, evading the Stadium.

In such an emergency situation, there may be various cases in which Connect can be of help. As described in [CON11c], the police could for example be directly put in connection with various surveillance systems in the zone to receive videos or pictures in their devices. We focus on the case that a policeman that sees the suspect running away can dynamically seek assistance to capture him from civilians serving as private security guards in the zone of interest. To get help in following the moves of the escaping terrorist and capturing him, the policeman sends to the civilian guards an alert message in which one picture of the suspect is distributed.

On their side, to perform their service, the guards are equipped with smart radio transmitters which run an *EmergencyCall* application. This transmission follows a two steps protocol. We assume in fact that the guards that control a zone are CONNECTed in groups, and that for each group there is a Commander on duty. The protocol followed in the *EmergencyCall* application is that a request message is first sent from the guards control center to the Commander. As soon as the Commander replies with an acknowledgement of receipt, a message with details of the emergency is forwarded to all security guards. On correct receipt of the alert, each guard's device automatically sends an ack to the control center.

The two applications, *SecuredFileSharing* and *EmergencyCall*, in this scenario represent the two Networked Systems, which are not a priori compatible; hence a CONNECTor bridging between the policeman device and the guard device must be deployed.

In the following we show the LTSs modelling the two applications above mentioned.

SecuredFileSharing

- The peer that initiates the communication (hereafter denominated the *coordinator*) sends a broadcast message (`selectArea`) to selected peers (the Police control center or policemen) operating in a specified area of interest. In the SecuredFileSharing application, the coordinator can be either the Police control center or a policeman.
- The selected peers reply with an `areaSelected` message.
- The coordinator sends an `uploadData` message to transmit confidential data to the selected peers.
- Each selected peer automatically notifies the coordinator with an `uploadSuccess` message when the data have been successfully received.

EmergencyCall

- The guards control center sends an `eReq` message to the commanders of the patrolling groups operating in a given area of interest.
- The commanders reply with an `eResp` message.
- The guards control center sends an `emergencyAlert` message to all guards of the patrolling groups; the message reports the alert details.
- Each guard's device automatically notifies the guards control center with an `eACK` message when the data has been successfully received and a timeout is triggered after a time interval if not all guards sends back the `eAck` message. The timeout represents the maximum time that the CONNECTor can wait for the `eAck` message from the guards.

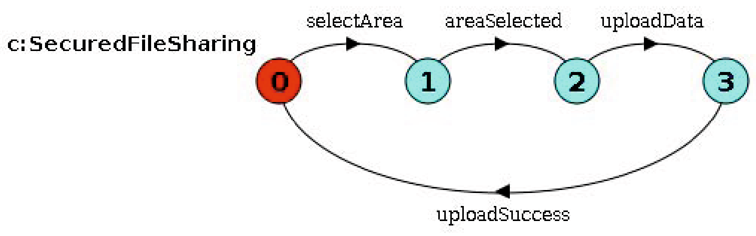
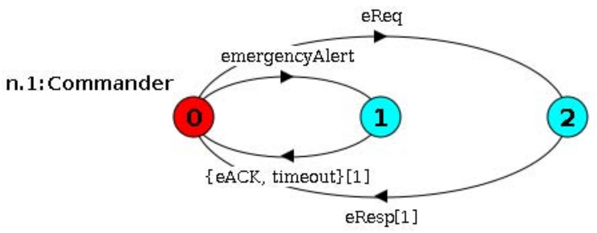
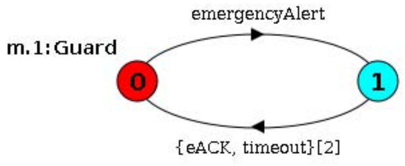


Fig. 13. LTS of the *SecuredFileSharing* Application



(a) Commander



(b) Other guards

Fig. 14. LTSs of the *EmergencyCall* Application

To allow a Policeman and the guards in the zone where the suspect has escaped to communicate we need to synthesize on-the-fly a CONNECTor. Precisely, we need to mediate between the LTSs shown in Figures 13 and 14, respectively. We briefly summarise the needed mappings below.

CONNECTor

– The `selectArea` message of the policeman is translated into an `eReq` message directed to the commander of the patrolling group operating in the area of interest.
– The `eResp` message of the commander is translated into an `areaSelected` message for the policeman.
– The `uploadData` message of the policeman is translated into a multicast `emergencyAlert` message.
– The `eACK` messages automatically sent by the guards' devices that correctly receive the `emergencyAlert` message are collected and then translated into a single `uploadSuccess` message for the policeman.

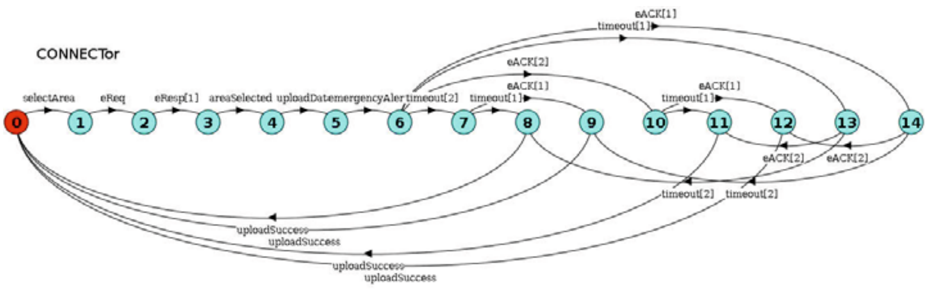


Fig. 15. LTS of the CONNECTor

The LTS of the CONNECTor in the case of a patrolling group consisting of one commander and two other guards is shown in Figure 15.

4.2 Off-line Analysis

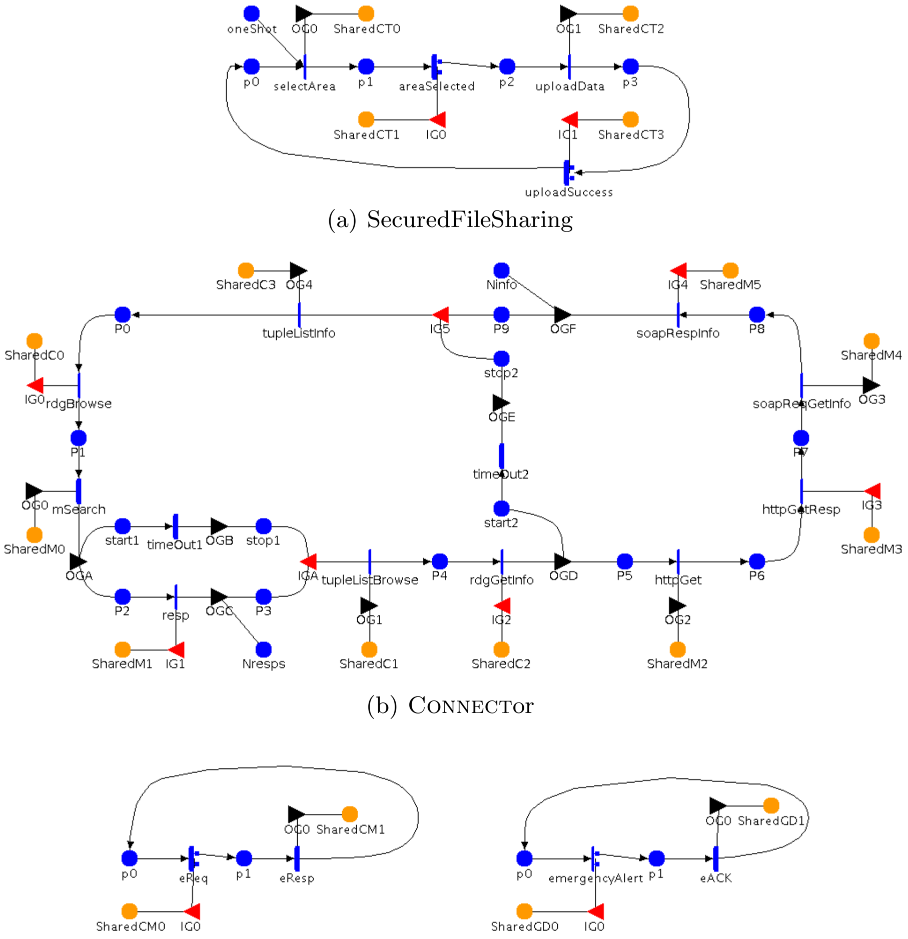
In this section, first we show the SAN models of the case study under analysis, then the results of the analysis obtained through Möbius.

SAN Models. The SAN models of guard, commander, CONNECTor, and SecuredFileSharing are shown in Figure 16. The model of the CONNECTed system is obtained by composing, via place sharing, the SAN models of SecuredFileSharing, commander, CONNECTor and guards (the SAN model of the guards is obtained by replicating a guard with the Rep operator). There is a shared place for each pair of activities that represent send/receive actions: send activities add tokens in the shared place, while receive activities remove tokens from the shared place and use the marking of the shared place as enabling condition. Note that, in general, a send activity may control $n > 1$ receive activities (e.g., in the case of a message with multicast/broadcast addresses); in this case, the send activity will add n tokens to the shared place to allow the simultaneous enabling of the receive activity of n receivers.

Timing aspects for send/receive actions are taken into account in the SAN models as follows: when n receive activities complete simultaneously after a send action completes, the receive activities are instantaneous and the send activity is timed; when n receive activities complete independently after a send action completes, the receive activities are timed and the send activity is instantaneous. Timeouts are modelled with timed activities that force the enabling of other activities.

In the following we describe in detail the behaviour of the model of CONNECTed system. In the description, we will use the prefixes `C`, `G`, `CON`, and `S` to disambiguate the names of local places, activities and gates of commander, guards, CONNECTor, and SecuredFileSharing.

Initially, all places in the models have zero tokens, except `p0`, which contains one token in all models. The SecuredFileSharing starts the communication, because `S.selectArea` is the only enabled activity. When `S.selectArea`



(a) SecuredFileSharing

(b) CONNECTor

(c) EmergencyCall, commander (d) EmergencyCall, guard

Fig. 16. SAN Models

completes, one token is placed in `S.p1` and one token in `SharedCT0`. At this point, `S.selectArea` is enabled. When `S.selectArea` completes, one token is placed in `S.p1` and the number of tokens in `SharedCT0` is increased. The activity `CON.selectArea` is now enabled, when it completes one token is moved from `SharedCT0` to `CON.p1`, and `CON.eReq` becomes enabled. When `CON.eReq` completes, the marking changes as follows: *commNum* tokens are placed in `SharedCM0`, because *commNum* commanders must be involved in the communication; *commNum* tokens are placed in `CON.p2`, because the CONNECTor must wait for one `eResp` from each commander. When the CONNECTor receives a response from each commanders, (i) for each response received one token is placed in `CON.p3`; (ii) when each commanders has sent a response `CON.areaSelected` is enabled, one token is placed in `CON.p4` and the number of tokens in `SharedCT1` is increased. At this point `S.areaSelected` is enabled, when it completes one token is moved from `SharedCT1` to `S.p2`, and `S.uploadData` becomes enabled. A token is placed in `S.p3` and the number of token in `SharedCT2` is increased. Activity `CON.uploadData` is now enabled, when it completes one token is moved from `SharedCT2` to `CON.p5`, which enables the activity `emergencyAlert`. When `emergencyAlert` completes $commNum + guardNum$ tokens are placed both in `SharedGD0` and `CON.p6`, and the number of tokens in `CON.start1` is increased. At this point activities `CON.timeOut1` and `G.emergencyAlert` are both enabled. The first one represents the CONNECTor's timeout on the maximum waiting time; while the second one enables the activity `G.eACK` which increases the number of tokens in `SharedGD1`. At this point activity `CON.eACK` is enabled and the number of tokens in `CON.p7` and `CON.Nresps` is increased, until the timed activity `CON.timeOut1` completes. The activity `uploadSuccess` becomes enabled when $commNum+guardNum$ tokens are placed in `CON.p7`, this means that the CONNECTor has received all responses, or when the number of tokens in `CON.stop1` is greater than zero, this means that the time associated to the activity `timeOut1` has elapsed, `CON.timeOut1` completes. The number of tokens in `CON.Nresps` represents the number of guards that have recived the `emergencyAlert` and have sent back the `eACK` before the timeout.

State-based Stochastic Analysis. The analysis performed through Möbius consists in: i) two measures of latency, at varying the number of guards and for different traffic patterns; ii) a measure of coverage in case of failure.

Latency. This property is measured from the moment when the control center starts to send the initial request `selectArea` to the time it receives `uploadSuccess`. The latency is specified by accumulating over time the following rate reward function:

```
double latency() {
    if ( SecuredFileSharing->p1->Mark() > 0
         || SecuredFileSharing->p2->Mark() > 0
         || SecuredFileSharing->p3->Mark() > 0 )
       { return 1; }
}
```

Latency2. It is also useful to know the trend of the amount of time spent on waiting for `eAck`, given different values of **T**, the duration of the timeout shown in the model of the CONNECTor,.

This property is specified by accumulating over time the following rate reward function:

```
double latency2() {
    if ( connector->start1->Mark() > 0 && connector->p6->Mark() > 0 )
       { return 1; }
}
```

Coverage. This property is associated to the real value m/n, where n represents the total number of guards and commanders, and m represents how many of them send back their respons to the connector within **T** time units, after they receive the request `emergencyAlert`.

Coverage is specified by accumulating over time the following impulse reward on `CON.uploadSuccess` (`guardNum` and `commNum` are two parameters of the composed model, and hold the number of guards and commanders respectively):

```
double coverage() {
 return ( (double) connector->Nresps->Mark() ) / ( guardNum + commNum );
}
```

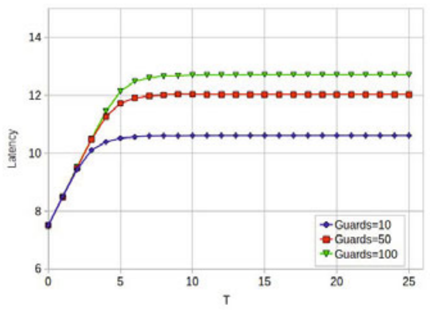
Connected systems may include an arbitrary large number of Networked Systems. Therefore, we investigated the scalability of the SAN model of the Connected system by analysing large networks. The developed SAN model of the Connected system is parametric with respect to the number of guards and commanders.

We successfully assessed coverage and latency for scenarios with hundreds of guards and two commanders. Figure 17(a) shows the analysis results for latency in scenarios with at most 100 guards. We can notice that, for low values of the timeout **T**, it is not possibile to appreciate differences in latency at increasing the number of guards. In fact, due to the short duration of **T** the guards do not have enough time to send a response. When **T** becomes greater than 8 time units, it is possible to observe how the number of guards affects the value of latency: as expected, increasing the number of guards leads to an increase of latency.

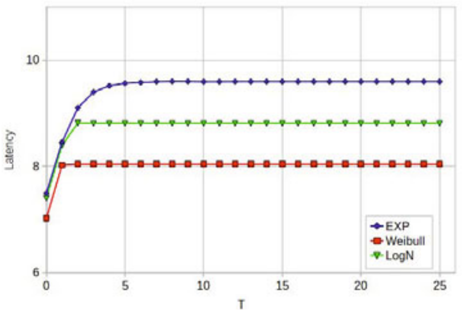
The number of batches needed to reach a confidence level of 95% and a confidence interval of 10% for the considered models was always below $10K$, because the models are relatively simple.

Latency for different traffic patterns. Connected systems are expected to be a mix of heterogeneous user applications, each of which may have different characteristics and requirements. Currently, there is no single traffic distribution that can efficiently capture the traffic characteristics of all types of networks under every possible situation. A large number of empirical studies have shown that network traffic is self-similar and that it generally exhibits multiple time-scale behaviour [LTWW94]. These aspects can be modelled with subexponential distributions, such as Weibull and Lognormal.

We investigated the effect of different subexponential distributions on latency by changing the probability distribution function of the timed activities. For a fair comparison, we have chosen distribution parameters that allow the same mean value in all cases. The analysis results are shown in Figure 17(b). We can notice that different traffic patterns lead to different latency profiles. Similarly to the previous analysis, the latency assumes a constant value when the timeout **T** reaches a certain value (5 time units in this case), after which it is possible to appreciate how different traffic patterns affect the latency.



(a) Latency for different number of guards



(b) Latency for different traffic patterns

Fig. 17. Latency for Different System Size and Different Traffic Patterns

Coverage in the case of failures. Communication in the real-world can be subject to failures. Therefore, failure modes need to be accounted for when setting up the system model. Failure modes can pertain the value domain (e.g., wrong output), and/or the time domain (e.g., omission). In this section, we assess coverage in the case of omission failure of the messages sent and received in the EmergencyCall application. Figure 18 shows the coverage profiles for different probability *P(ECallFailure)* of failures of `EmergencyCall` communications. The analysis is performed with two commanders and two guards. The figure shows that variations in the failure probability significantly affect the coverage metric. The lower values shown by all the curves on the left side of the figure (that is, at initial values of **T**) are due to the fact that, given the short duration of **T**, the guards do not have enough time to send a response.

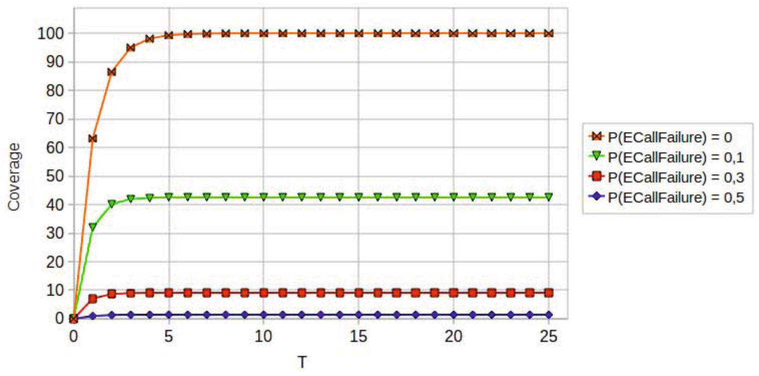


Fig. 18. Coverage for Different *P(ECallFailure)* of Failures of `EmergencyCall` Communications

4.3 On-line Analysis

In the following, we focus on the Enablers interactions only, leaving out of scope the actions taken by DePer Enabler once it obtains the values observed at run-time from the Glimpse Enabler. We show a basic interaction between DePer and Glimpse Enablers, with reference to the Terrorist Alert Scenario [CON11c]. As summarised in Section 4.1, the scenario considers the interactions between police and patrolling civil guards. It is assumed that policemen and guards are both equipped with mobile devices, but they use different communication protocols. Hence, we intend to use the Connect infrastructure to enable the direct interoperation between a policeman and guards in the zone.

What we want to monitor at run-time is the latency between two states of the LTS. Specifically, we want to monitor latency of two transitions from the LTS shown in Figure 13.

The parameters under monitoring are the duration of the transitions executed by the NS requesting the communication, on which timeouts have been setup in the Connector specification to limit the waiting periods. Therefore, having feedbacks on real executions is useful to improve the timeout calibration.

From Figure 13, the events to be monitored are the consumer transitions `selectArea` and `areaSelected`. The request messages sent by DePer Enabler to Glimpse are shown in Listing 2.

The Glimpse infrastructure, more specifically, the Manager component, receives the DePer requests and sets up the ComplexEventProcessor with the provided rule.

The events flowing in from Probes are structured on a `ConnectBaseEvent` object (see Figure 9), that provides all the necessary informations for an accurate pattern recognition.

According to the scenario, the peer that initiates the communication sends a broadcast message `selectArea` to selected peers (the Police control center or policemen) operating in a specified area of interest.

The event generated from the Probe instrumented into the peer software component is shown in Figure 19 and flows in into the Glimpse infrastructure stream of events.

When the selected peer replies (Police control center or policemen), another event is fired and sent on the Monitoring Bus, the `areaSelected` event.

The rule `computation time`, (lines 15-20) in Listing:2, used the timestamp contained into the two different events, matching: connectorID, sequenceID, ConnectorInstanceID, ConnectorInstanceExecutionID of each event. This rule, is able to calculate the latency (line 27) and provide it to the DePer.

Indeed, the rule `pending request` in the Listing2, (lines 40-42), computes the number of incoming requests into the Connector and provide it to DePer in order to evaluate coverage metric.

With those results, DePer is able now to evaluate the behaviour of the Connector and if this is not compliant to the expected values, it may contact the Syntesis Enabler requiring a new synthesis process.

Using a CEP able to infer more complex rules and patterns along with an event-driven architecture approach for dependability and performance analysis, may be beneficial in order to provide a cross-checking validation between run-time value and analysis expected value.

5 Related Work

This work spans over automated model-based dependability analysis and event-based monitoring.

Research on definition and development of transformation-based verification and validation environments are being pursued since several years. Providing automatic/automated transformations methods from system specification languages to modelling languages amenable to perform dependability analysis has been recognized as an important support for improving the quality of systems. In addition, it favours the application of verification and validation techniques at industry level, where these methods are not widely used primarily due to the high level of abstractness of the mathematical modelling and analysis techniques. To provide some examples, the Viatra tool [CHM$^+$02] automatically checks consistency, completeness, and dependability requirements of systems designed using the Unified Modeling Language. The Genet tool [CCK09], based on the theory of regions [ER90], allows the derivation of a general Petri net from a state-based representation of a system. Our work addresses the transformation from the LTS formalism, as system specification language, to SAN, as dependability modelling language. Since there are some steps in common with the Genet tool and related theory, we partially reused available results from this previous study.

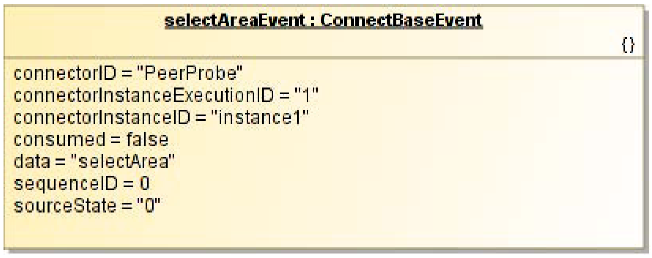
Similarly to Glimpse, also [PSB04] presents an extended event-based middleware with complex event processing capabilities on distributed systems. Similar to Glimpse this work adopts a publish/subscribe infrastructure but it is mainly focused on the definition of a complex-event specification language. The aim of Glimpse is to give a more general and flexible monitoring infrastructure for achieving a better interpretability with many possible heterogeneous systems.

Another monitoring architecture for distributed systems management is presented in [HAwM99]. Differently from Glimpse, this architecture employs a hierarchical and layered event filtering approach. The goal of the authors is to improve monitoring scalability and performance for large-scale distributed systems, minimizing the monitoring intrusiveness.

Many works focus on the definition of expressive complex event specification languages [MSS97, CM94, CM10]. Among these languages, GEM [MSS97] is a generalized and interpreted event monitoring language. It is rule-based (similar to other event-condition-action approaches) and also provides a tree-bases detection algorithm taking into account communication delay. Also the Snoop language [CM94] follows an event-condition-action approach supporting temporal and composite events specification but it is especially developed for active databases. A more recent formally defined specification language is TESLA [CM10]. It has a simple syntax and a semantics based on a first order

```
<?xml version="1.0" encoding="UTF-8"?>
<ComplexEventRuleActionList xmlns="http://labse.isti.cnr.it
    /glimpse/xml/ComplexEventRule"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://labse.isti.cnr.it/glimpse/xml/
     ComplexEventRule ./ComplexEventRule.xsd">
     <Insert RuleType="drools">
          <RuleName>transitionDurationRule</RuleName>
          <RuleBody>
     [...]
     rule "computation time"
     no-loop
     salience 999
     dialect "java"
     when
        $aEvent : SimpleEvent(this.data == "selectArea", this
             .getConsumed == false);
        $bEvent : SimpleEvent(this.data == "areaSelected",
                        this.getConsumed == false ,
                        this.getConnectorID == $aEvent.
                             getConnectorID ,
                        this.getConnectorInstanceID == $aEvent.
                             getConnectorInstanceID ,
                        this.getConnectorInstanceExecutionID ==
                                                   $aEvent.
                             getConnectorInstanceExecutionID ,
                        this after $aEvent);
     then
        $aEvent.setConsumed(true);
        $bEvent.setConsumed(true);
        SatisfiedRequest sr = new SatisfiedRequest();
        sr.setIncoming($aEvent);
        sr.setOutcoming($bEvent);
        sr.setDuration(DroolsUtils.latency($aEvent.
             getTimestamp(),$bEvent.getTimestamp()));
        insert(sr);
        retract($aEvent);
        retract($bEvent);
        ResponseDispatcher.NotifyMe(drools.getRule().getName
             (),"DePer Module", sr.getDuration());
     end

     rule "pending request"
     no-loop
     salience 999
     dialect "java"
     when
        $total : Number()
        from accumulate($nEvent : SimpleEvent(data == "
             selectArea")
              from entry-point "DEFAULT",
              count($nEvent))
     then
        ResponseDispatcher.NotifyMe(drools.getRule().getName
             (),"DePer Module", "PENDING: " + $total);
     end
          </RuleBody>
     </Insert>
</ComplexEventRuleActionList>
```

Listing 2. Sample Request from Dependability&Performance Enabler



selectAreaEvent : ConnectBaseEvent

{}

connectorID = "PeerProbe"
connectorInstanceExecutionID = "1"
connectorInstanceID = "instance1"
consumed = false
data = "selectArea"
sequenceID = 0
sourceState = "0"

Fig. 19. The selectArea Event Sent from Peer Probe

temporal logic. The authors of [CM10] also provide an efficient event detection algorithm by translating TESLA rules into automata. Some existing open-source event processing engines are Drools Fusion [dro] and Esper [esp]. They can fully be embedded in existing Java architectures and provide efficient rule processing mechanisms. In our prototype we used Drools because ServiceMix offers it as business rule engine.

Preliminary studies that attempt combining off-line with on-line analysis have already appeared in the literature. A major area on which such approaches have been based is that of autonomic computing. Among such studies, in [MT06], an approach is proposed for autonomic systems, which combines analytic availability models and monitoring. The analytic model provides the behavioural abstraction of components/subsystems and of their interconnections and dependencies, while statistical inference is applied on the data from real time monitoring of those components and subsystems, to assess parameter values of the system availability model. Through on-line monitoring and estimation of system availability, adaptive on-line control of system availability can then be obtained. In [RP10], an approach is proposed to carry out run-time reliability estimation, based on a preliminary modelling phase followed by a refinement phase, where real operational data are used to overcome potential errors due to model simplifications. The model is based on Discrete Time Markov Chain, and a prototype version of the monitoring system has been implemented, that is initially trained with the reference model and the preliminary reliability estimation, and then uses operational data to compute the on-line reliability level.

Our approach aims at proposing powerful evaluation and monitoring supports able to cover, individually, a wide spectrum of needs inside the Connect framework (quantitative assessment of a variety of dependability and performance metrics on one side and generic monitoring infrastructure useful to a variety of Connect Enablers on the other side), and at exploiting their synergic usage to lead to higher accuracy of dependability and performance analysis results.

6 Conclusions and Outlook

We have presented the directions currently pursued in the Connect project for the assessment of dependability and performance related properties of dynamic evolving systems. In particular, we focused on usage of stochastic model-based approaches, both at design time, for the early evaluation of the relevant non-functional requirements, and at run-time, for the continuous checking of system behaviour based on the actual data collected by the publish-subscribe monitoring infrastructure.

In line with the tutorial flavour of the chapter, we first provided basic introductory concepts and bibliography to model-based analysis of dependability attributes, relying on the SAN formalism and the Möbius tool. We also overviewed event-based monitoring and current research directions. We then described the solutions developed in the Connect project, which include the DePer modular infrastructure and the flexible Glimpse monitor, and discussed their interconnection to bring dependability and performance analysis to on-line stage.

The presented solutions mostly exploit advanced state-of-art results. The value brought forward by the Connect project stays mainly in their combined engineering and in the integration with the other Connect Enablers. The infrastructure resulting from the interfacing of DePer and Glimpse has been conceived with the highest flexibility and modularity in mind, so to allow for future further expansions, for example by including differing analysis engines, as we already show in [DGKM+10] for stochastic model checking.

At the time of writing, the implementation of the presented framework is still on-going and therefore our future work in the short term will of course involve the experimentation and refinement of the proposed approaches. More importantly, we intend to make the framework *model-driven*, so to make it more general and reusable. We are defining a property meta-model, a first release of which is available at http://labsewiki.isti.cnr.it/labse/tools/cpmm/public/main. The meta-model specifies non-functional properties, both qualitative and quantitative, to be evaluated. The idea then is that specific property models conforming to such meta-model can be used to automatically drive both DePer analysis, by providing in input the requested dependability and performance metrics, and probe instrumentation of the Connect monitoring Enabler.

Acknowledgements

The work reported in this chapter has been partially supported by the EU-funded Connect project (FP7231167) and stems from the collaborative effort of many colleagues from the Connect project, whom we would like to thank collectively, in the difficulty to list each single contribution. Among them, however, a special acknowledgement goes to the contributions from Marco Martinucci, Paolo Masci and Antonino Sabetta, formerly part of the CNR-ISTI Connect team, who have conceived with us the architectures of the presented DePer and Glimpse infrastructures, contributed to their development, and shared with us lot of discussions, insights, writing, and perspectives. We and the project owe much to them for the presented achievements.

References

[ABC84] Ajmone Marsan, M., Balbo, G., Conte, G.: A class of generalized stochastic petri nets for the performance evaluation of multiprocessor systems. ACM Transactions on Computer Systems 2(2), 93–122 (1984)

[AC87] Ajmone Marsan, M., Chiola, G.: On Petri nets with deterministic and exponentially distributed firing times. In: Rozenberg, G. (ed.) APN 1987. LNCS, vol. 266, pp. 132–145. Springer, Heidelberg (1987)

[ALRL04] Avizienis, A., Laprie, J.-C., Randell, B., Landwehr, C.: Basic concepts and taxonomy of dependable and secure computing. IEEE Transactions on Dependable and Secure Computing 1(1), 11–33 (2004)

[Bal01] Balbo, G.: Introduction to stochastic petri nets. In: Katoen, J.-P., Brinksma, H., Hermanns, H. (eds.) EEF School 2000 and FMPA 2000. LNCS, vol. 2090, pp. 84–155. Springer, Heidelberg (2001)

[BCDG$^+$ar] Bertolino, A., Calabrò, A., Di Giandomenico, F., Martinucci, M., Masci, P.: Automated refinement of dependability analysis through monitoring in dynamically connected systems. In: Proc. IEEE International Symposium on Autonomous Decentralized Systems, Tokyo, Japan (March 2011, to appear)

[BCG05] Bondavalli, A., Chiaradonna, S., Di Giandomenico, F.: Model-based evaluation as a support to the design of dependable systems. In: Diab, H.B., Zomaya, A.Y. (eds.) Dependable Computing Systems: Paradigms, Performance Issues, and Applications, pp. 57–86. Wiley, Chichester (2005)

[BGD06] Baresi, L., Ghezzi, C., Di Nitto, E.: Toward open-world software: issues and challenges. Computer 39(10) (2006)

[BGG04] Baresi, L., Ghezzi, C., Guinea, S.: Smart monitors for composed services. In: ICSOC 2004: Proceedings of the 2nd International Conference on Service Oriented Computing, pp. 193–202. ACM, New York (2004)

[BPTT98] Bobbio, A., Puliafito, A., Telek, M., Trivedi, K.S.: Recent developments in non-Markovian stochastic Petri nets. Journal of Circuits, Systems and Computers 8(1), 119–158 (1998)

[BS02] Buy, U.A., Singal, G.: Toward efficient algorithms for generating compact petri nets from labeled transition systems. In: COMPSAC 2002, pp. 717–722. IEEE Computer Society, Washington, DC, USA (2002)

[BT98] Bobbio, A., Telek, M.: Non-exponential stochastic Petri nets: an overview of methods and techniques. Computer Systems Science and Engineering 13(6), 339–351 (1998)

[CBC$^+$93] Ciardo, G., Blakemore, A., Chimento, P.F., Muppala, J.K., Trivedi, K.S.: Automated generation and analysis of markov reward models using stochastic reward nets. In: Meyer, C., Plemmons, R.J. (eds.) Linear Algebra, Markov Chains, and Queueing Models. IMA Volumes in Mathematics and its Applications, vol. 48, pp. 145–191. Springer, Heidelberg (1993)

[CCD$^+$01] Clark, G., Courtney, T., Daly, D., Deavours, D.D., Derisavi, S., Doyle, J.M., Sanders, W.H., Webster, P.G.: The Mobius modeling tool. In: 9th Int. Workshop on Petri Nets and Performance Models, Aachen, Germany, pp. 241–250. IEEE Computer Society Press, Los Alamitos (2001)

[CCK09] Carmona, J., Cortadella, J., Kishinevsky, M.: Genet: A tool for the synthesis and mining of petri nets. In: ACSD 2009, pp. 181–185. IEEE Computer Society, Washington, DC, USA (2009)

[CGL94] Ciardo, G., German, R., Lindemann, C.: A characterization of the stochastic process underlying a stochastic petri net. IEEE Transactions on Software Engineering 20(7), 506–515 (1994)

[CHM$^+$02] Csertan, G., Huszerl, G., Majzik, I., Pap, Z., Pataricza, A., Varro, D., Varr, D.: Viatra - visual automated transformations for formal verification and validation of uml models. In: 17th IEEE International Conference on Automated Software Engineering (ASE 2002), pp. 267–270 (2002)

[CKLY98] Cortadella, J., Kishinevsky, M., Lavagno, L., Yakovlev, A.: Deriving petri nets from finite transition systems. IEEE Transactions on Computers 47(8), 859–882 (1998)

[CKT94] Choi, H., Kulkarni, V.G., Trivedi, K.S.: Performance modeling using Markov regenerative stochastic Petri nets. Performance Evaluation 20(1-3), 339–356 (1994)

[CM94] Chakravarthy, S., Mishra, D.: Snoop: An expressive event specification language for active databases. Data & Knowledge Engineering 14(1), 1–26 (1994)

[CM10] Cugola, G., Margara, A.: TESLA: a formally defined event specification language. In: Proceedings of DEBS, pp. 50–61 (2010)

[CON10] CONNECT Consortium. Deliverable 5.1 – Conceptual Models for Assessment & Assurance of Dependability, Security and Privacy in the Eternal CONNECTed World (2010)

[CON11a] CONNECT Consortium. Deliverable 4.2 – Further development of learning techniques (2011)

[CON11b] CONNECT Consortium. Deliverable 5.2 – Design of Approaches for Dependability and Initial Prototypes (2011)

[CON11c] CONNECT Consortium. Deliverable 6.1 – Experiment scenarios, prototypes and report Iteration 1 (2011)

[CON13] EU FP7 Project CONNECT (FP7–231167) (2009-2013)

[DGKM$^+$10] Di Giandomenico, F., Kwiatkowska, M., Martinucci, M., Masci, P., Qu, H.: Dependability analysis and verification for CONNECTed systems. In: Margaria, T., Steffen, B. (eds.) ISoLA 2010. LNCS, vol. 6416, pp. 263–277. Springer, Heidelberg (2010)

[dro] Drools fusion: Complex event processor, http://www.jboss.org/drools/drools-fusion.html

[ER90] Ehrenfeucht, A., Rozenberg, G.: Partial (set) 2-structures. Part I: basic notions and the representation problem. Acta Inf. 27(4), 315–342 (1990)

[esp] Esper: Event stream and complex event processing for java, http://www.espertech.com/products/esper.php

[Fid96] Fidge, C.: Fundamentals of Distributed System Observation. IEEE Softw. 13(6), 77–83 (1996)

[Gai86] Gait, J.: A Probe Effect in Concurrent Programs. Softw., Pract. Exper. 16(3), 225–233 (1986)

[Ger01] German, R.: Non-Markovian analysis. In: Brinksma, E., Hermanns, H., Katoen, J.P. (eds.) EEF School 2000 and FMPA 2000. LNCS, vol. 2090, pp. 156–182. Springer, Heidelberg (2001)

[GGB$^+$11] Grace, P., Georgantas, N., Bennaceur, A., Blair, G., Chauvel, F., Issarny, V., Paolucci, M., Saadi, R., Souville, B., Sykes, D.: The CONNECT architecture. In: Bernardo, M., Issarny, V. (eds.) SFM 2011. LNCS, vol. 6659, pp. 350–392. Springer, Heidelberg (2011)

[Hav01] Haverkort, B.R.: Markovian models for performance and dependability evaluation. In: Katoen, J.-P., Brinksma, H., Hermanns, H. (eds.) EEF School 2000 and FMPA 2000. LNCS, vol. 2090, pp. 38–83. Springer, Heidelberg (2001)

[HAwM99] Hussein, E.A.-S., Abdel-wahab, H., Maly, K.: HiFi: A New Monitoring Architecture for Distributed Systems Management. In: Proceedings of ICDCS, pp. 171–178 (1999)

[HBPU06] Hallal, H., Boroday, S., Petrenko, A., Ulrich, A.: A formal approach to property testing in causally consistent distributed traces. Formal Asp. Comput. 18(1), 63–83 (2006)

[How71] Howard, R.A.: Dynamic Probabilistic Systems: Markov Models. Decision and Control, vol. 1. John Wiley and Sons, New York (1971)

[iee90] IEEE Std 610.12-1990: IEEE Standard Glossary of Software Engineering Terminology (1990)

[jbi] Jbi: Java business integration, `http://jcp.org/aboutJava/communityprocess/final/jsr208`

[JLSU87] Joyce, J., Lomow, G., Slind, K., Unger, B.: Monitoring distributed systems. ACM Trans. Comput. Syst. 5(2), 121–150 (1987)

[Lam78] Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM 21(7), 558–565 (1978)

[Lap95] Laprie, J.C.: Dependable computing and fault tolerance: concepts and terminology. In: Twenty-Fifth International Symposium on Fault-Tolerant Computing, 1995, Highlights from Twenty-Five Years, pages 2+ (1995)

[Lap08] Laprie, J.C.: From dependability to resilience. In: 38th IEEE/IFIP Int. Conf. on Dependable Systems and Networks (2008)

[LTWW94] Leland, W.E., Taqqu, M.S., Willinger, W., Wilson, D.V.: On the self-similar nature of ethernet traffic (extended version). IEEE/ACM Transactions on Networking 2(1), 1–15 (1994)

[MCC04] Massie, M.L., Chun, B.N., Culler, D.E.: The ganglia distributed monitoring system: design, implementation, and experience. Parallel Computing 30(7), 817 (2004)

[Mey92] Meyer, J.F.: Performability: A retrospective and some pointers to the future. Perform. Eval. 14(3-4), 139–156 (1992)

[MFT00] Muppala, J.K., Fricks, R.M., Trivedi, K.S.: Techniques for system dependability evaluation. In: Grassmann, W.K. (ed.) Computational Probability. Operations Research and Management Science, vol. 24, pp. 445–480. Kluwer Academic Publishers, The Netherlands (2000)

[MK06] Magee, J., Kramer, J.: Concurrency: state models & Java programs. John Wiley & Sons, New York (2006)

[MM84] Movaghar, A., Meyer, J.F.: Performability modelling with stochastic activity networks. In: 1984 Real-Time Systems Symposium, Austin, TX, pp. 215–224. IEEE Computer Society Press, Los Alamitos (December 1984)

[MMDGar] Masci, P., Martinucci, M., Di Giandomenico, F.: Towards automated dependability analysis of dynamically connected systems. In: Proc. IEEE International Symposium on Autonomous Decentralized Systems. IEEE, Tokyo (March 2011, to appear)

[Mol82] Molloy, M.K.: Performance analysis using stochastic Petri nets. IEEE Transactions on Computers 31(9), 913–917 (1982)

[MSS94] Mansouri-Samani, M., Sloman, M.: Monitoring distributed systems. pp. 303–347 (1994)

[MSS97] Mansouri-Samani, M., Sloman, M.: GEM: a generalized event monitoring language for distributed systems. Distributed Systems Engineering 4(2), 96–108 (1997)

[MT06] Mishra, K., Trivedi, K.S.: Model based approach for autonomic availability management. In: Penkler, D., Reitenspiess, M., Tam, F. (eds.) ISAS 2006. LNCS, vol. 4328, pp. 1–16. Springer, Heidelberg (2006)

[NST04] Nicol, D.M., Sanders, W.H., Trivedi, K.S.: Model-based evaluation: from dependability to security. IEEE Transactions on Dependable and Secure Computing 1, 48–65 (2004)

[PSB04] Pietzuch, P.R., Shand, B., Bacon, J.: Composite event detection as a generic middleware extension. IEEE Network 18(1), 44–55 (2004)

[ReS08] ReSIST Consortium. EU project ReSIST: Resilience for Survivability in IST. Deliverable D33: Resilience-explicit computing. Technical report (2008), http://www.resist-noe.org/

[RP10] Trivedi, K.S., Pietrantuono, R., Russo, S.: Online monitoring of software system reliability. In: Proc. EDCC 2010 - 2010 European Dependable Computing Conference, pp. 209–218. IEEE Computer Society, Los Alamitos (2010)

[rpc] RPC: Model for programming in a distributed computing environment, http://msdn.microsoft.com/enus/library/ms691207(VS.85).aspx

[RSB05] Raffelt, H., Steffen, B., Berg, T.: Learnlib: a library for automata learning and experimentation. In: FMICS 2005, pp. 62–71. ACM, New York (2005)

[rul] Ruleml: The rule markup initiative, http://ruleml.org

[Sch95] Schroeder, B.A.: On-Line Monitoring: A Tutorial. Computer 28(6), 72–78 (1995)

[Sha93] Shah, B.P.: Analytic solution of stochastic activity networks with exponential and deterministic activities. Master's thesis, University of Arizona, USA (1993)

[SI10] Spalazzese, R., Inverardi, P.: Mediating connector patterns for components interoperability. In: Babar, M.A., Gorton, I. (eds.) ECSA 2010. LNCS, vol. 6285, pp. 335–343. Springer, Heidelberg (2010)

[SK88] Spezialetti, M., Kearns, J.P.: A General Approach to Recognizing Event Occurences in Distributed Computations. In: ICDCS, pp. 300–307 (1988)

[SM02] Sanders, W.H., Meyer, J.F.: Stochastic Activity Networks: formal definitions and concepts, pp. 315–343 (2002)

[Tri02] Trivedi, K.S.: Probability and Statistics with Reliability, Queueing and Computer Science Applications, 2nd edn. John Wiley & Sons, New York (2002)

[Zim99] Zimmer, D.: On the semantics of complex events in active database management systems. In: Proceedings of the 15th International Conference on Data Engineering, p. 392. IEEE Computer Society, Washington (1999)

Security and Trust*

Gabriele Costa[1,2], Valérie Issarny[3], Fabio Martinelli[1], Ilaria Matteucci[1], and Rachid Saadi[3]

[1] IIT CNR, Pisa, Italy
[2] Università di Pisa, Italy
[3] INRIA, France
{gabriele.costa,fabio.martinelli,ilaria.matteucci}@iit.cnr.it,
{valerie.issarny,rachid.saadi}@inria.fr

Abstract. Security and Trust offer two different prospectives on the problem of the correct interaction among software components. For many aspects, they represent complementary viewpoints. Moreover, in the study of the verification of non-functional properties of programs they represent a mainstream. Several security aspects, *e.g.*, access control, could be based also on trust and, vice versa, trust models could update the level of trust of a (component of a) system according to the satisfaction of a particular security policies. According to that, here we present the *Security-by-Contract-with-Trust* framework, $S\times C\times T$ for short. It has been developed considering a system platform that has to execute an application whose developer is unknown in such a way that security policies set on it are not violated. The $S\times C\times T$ mechanism is driven by both security and trust aspects. It is based of three main concepts: the *application code*, the *application contract*, and the *system security policy* The level of trust we consider measures the adherence of the application code to its contract, *i.e.*, if the code respects its contract then the application is trusted, otherwise its level of trust decreases. According to the level of trust of the application, $S\times C\times T$ decides if check the contract against the policies and if the answer is positive, execute the application just monitoring its contract, or directly enforce the security policy set on the platform.

In order to better describe how the proposed mechanism works, we present its application to a mobile application marketplace scenarios. In this way we are also able to show its possible advantages in terms of performances and modularity.

Keywords: Security properties, Trust model, Managing of trust feedback, Run-time enforcement, Contract monitoring.

1 Introduction

In the last decades, the number of devices, *e.g.*, mobile phones, smart phones and slates, used in our daily life has been rapidly growing up. Furthermore, the computational capabilities of such devices tend to increase over and over. In practice, they combine the fair hardware profiles, *e.g.*, CPU, with high level connectivity. Hence, they can download and run a rich variety of quite complex applications.

* Work partially supported by EU-funded project FP7-231167 CONNECT and by EU-funded project FP7-256980 NESSoS.

Mobile Java applications (MIDlets) offer a clear example of fixed trust relationship. Indeed, a MIDlet is a software released by some vendor that clients download and install on their device. The potential constraints on the resources of mobile devices (*e.g.*, battery and memory) make several security mechanisms practically infeasible. The current technique for providing security assurances to mobile device users is based on software certification released by a accredited *certification authority* (CA). However, the certificate-based approach has several, well known drawbacks. Mainly, it implements a white list strategy. While certified MIDlets have all the privileges they need, uncertified applications have very little access to the system independently from their actual behaviour, leading to a significant reduction of their usability. On the other hand, executing a malicious, signed application can have obvious, dramatic consequences. There are many ways in which this attack can take place. A simple attacking scenario is based on the user's unawareness about security. Basically, a device owner wanting to install a MIDlet could decide to ignore whether it is not signed. This scenario is becoming popular, for instance, with local providers offering small, contextual applications (*e.g.*, catalogues, interactive guides). Often, MIDlet spots dispatch unsigned or self-certified applications to users moving inside some area of interest (*e.g.*, a museum).

Another danger arises from the hierarchical structure of certificates. In fact, when purchasing a certificate, the owner is often authorized to produce and distribute sub-certificates. The features of a sub-certificate depend on the structure of the original one (*e.g.*, a certificate can generate sub-certificates with an expiration date lower or equal to its own). For instance, an attacker acquiring a certificate can use it for signing a malicious MIDlet. Then, after detecting the attack, it should be possible, analysing the certificate, to trace back the certificate history and discover what went wrong in the sub-certificates chain. However, this is a reactive approach that can lead to identifying misbehaving entities (CAs, developers, vendors), while, in general, a proactive solution would be preferable.

For overcoming these limitations, *contract-based* approaches have been presented. In these models contracts are in charge of providing guarantees on the correct behaviour of programs. Contracts are automatically produced by any provider and attached to the code. The counterpart of contracts are user-defined, security policies which define the safe behaviours, *i.e.*, those complying with the user's requirements. Hereafter we recall the Security-by-Contract (S$\times$C) paradigm that, among the others, received major attention.

Along this research direction, we also present the Security-by-Contract-with-Trust (S$\times$C$\times$T) framework. Briefly, S$\times$C$\times$T extends the previous approach by including a notion of trust. In particular, this process is implemented through a contract monitoring framework responsible for verifying whether a running application respects its contract. When an attack, namely an attempt to violate a contract, is detected, our system reacts immediately by enforcing a security policy and preventing the attack from being actually performed. Moreover, a contract breaking causes an automatic modification of the trust relationship between the device and the authority providing the contract. Informally, trust feedback are managed according to the concept of *mobile application criticality*. The degree of criticality reflects how much an application may be considered critical with respect to security and trust aspects according to its type and category (*i.e.*, according to which kind of data it may access or which resources it uses). Hence, this system can immediately react to threats and prevent further attacks coming for the same source.

In order to show how the proposed framework works, we present an application of the S×C×T to a Mobile Applications Marketplaces (MAMp) scenario, like, for instance, Apple AppStore, Cydia, Android Market, in which mobile applications (MA) are released by some vendors and customers can download and install them on their devices. The S×C×T mechanism manages trust by rewarding and penalising MA's provider according to the MA's trust recommendation given by the MAMp and the MA criticality. Furthermore, the proposed framework offers a high degree of flexibility providing applications clients with a reliability feedback and assuring security guarantees also under pervasive, contextual mobility conditions.

This chapter is structured as follows: Section 2 presents some background about security for mobile applications. Section 3 recalls background notions about trust management. Section 4 presents the Security-by-Contract-with-Trust framework starting by recalling the original Security-by-Contract paradigm and underlining the extensions and the differences. In Section 5 we show the application of the Security-by-Contract-with-Trust to the case study of a mobile application marketplace and in Section 6 we provide the conclusion of the chapter and some future directions.

2 Mobile Code Security

Traditionally, security solutions aim at protecting our resources from malicious entities by restricting and filtering their access. However, with the growth of mobile applications, the problem of securing critical resources got more complex [1]. Moreover, the recent success of the component-based development paradigms, *e.g.*, plug-in architectures for web browsers or mobile phones using micro applications, have made the mobile pieces of software even more common. Under these assumptions, mechanisms for defining and applying fine-grained security rules are needed for guaranteeing the correct behaviour and interactions of these systems.

In the last years, *history-based security* has received major attention from the researchers. Mainly, this is due to the model scalability and applicability. As a matter of fact, almost every running program performing security-relevant operations can be seen as the source of a stream of security actions. Security actions are the side effect of standard computation and their sequences, namely *execution traces* or *histories*, provide a precise characterisation of programs behaviour.

Histories can be observed action by action while they are produced, issued by analysing the instructions of a program or simply declared by the software provider. In general, we can classify the history-based security mechanisms according to the source of the traces they use and the moment in the application life-cycle in which they applies. As we consider the security on client-side, we identify two stages for the security analysis: *deploy-time* and *run-time*. Below we provide an overview of some of the most studied approaches in both the previous categories.

2.1 Deploy-Time Security Analysis

Some of the advantages obtained by analysing a program before actually executing it are straightforward. Basically, one can directly discharge faulty components without risking to execute dangerous code. Nevertheless, static analysis may be expensive in terms of system resources, *e.g.*, CPU and memory, and it could be out of the scope of limited capabilities devices, *e.g.*, mobile phones. In general, the cost of performing this kind of analysis varies according to the complexity of the property that one is interested to check.

For instance, *cryptographic code-signing* is used for certifying the source, *i.e.*, the producer, of a mobile application and its integrity. Checking the signature of an application is equivalent to verifying that the code has not been tampered or modified before reaching the customer's system. Typically, the signature process is implemented by means of private/public encryption of the binary code. Even though this analysis is fast enough to be usable on small devices, checking the code signature provides no guarantees on the actual safety of running it.

Type checking can offer some more advantages. Indeed, well-typed programs are proved to be free from several run-time errors, *e.g.*, buffer overflows. Hence, type checking an application guarantees useful security properties. For instance, a well-typed program never accesses memory slots it is not allowed to. Again, type checking is efficiently implemented in many systems. Nevertheless, these properties are far from being the totality of the safe behaviours that one wants to force.

The *Proof-Carrying Code* (PCC) approach [2] enables safe execution of code from untrusted sources by requiring a producer to furnish a proof regarding the safety of its application. The code costumer uses a proof validator to check that the proof is valid and then the foreign code can be safely executed. Although optimised proof checkers have been proposed, this approach relies on a quite strong assumption, that is the code producer is always aware about the security requirements of the consumes. However, in many cases it seems to be not very reasonable and, in particular, when the customers can modify their security policies.

Strongly inspired to PCC, the *Model-Carrying Code* (MCC) [3] approach is also based on the idea that untrusted code is accompanied by additional information, capturing the *security-relevant behaviour* of code. Then, this extra information is extracted, verified, *e.g.*, through signature checking, and finally compared with the customer's security requirements. In this way, the code consumers can verify their own security policies, but the verification process is usually time consuming or even cumbersome. Moreover, as the code must carry its own model, the size of the applications may sensibly increase.

2.2 Run-Time Security

Run-time enforcement is a common practice in systems security. The basic idea behind policy enforcement is to guide safe behaviours through a monitoring agent. The security monitor holds the system policy and is responsible for guaranteeing its validity in time. Whenever the monitor observes security-relevant actions, it checks whether they are allowed by the policy. If it is the case, the action source can go on with its operations. Instead, if a violation is detected, the monitor prevents it and reacts according to the security rules of the policy. Several reaction strategies exist.

Java [4] offers a clear example. Indeed, Java applications need an interpreter, namely a *Java virtual machine* (JVM), to be actually executed. Hence, a monitor controlling the JVM can work directly on the instructions flow of Java applications. Whenever a security violation occurs, the monitor can force the JVM to quit the current execution, skip the last operation, invoke some emergency function, throw an exception, etc.

The capabilities of a security monitor mainly depend on two factors: the monitor effectiveness and the policy expressiveness. For being effective a monitor must be non by-passable, *i.e.*, it is triggered every time a security action is taking place, and proactive, *i.e.*, it can prevent an action from being actually executed. Policy expressiveness is a consequence of the used language. In the last decades two formalisation of security automata have received major attention.

Schneider's automata [5] are slightly different from *non-deterministic finite state automata* (NFA). Basically, they are defined by a finite set of states S, an alphabet A of observable actions and a set of labelled transitions $T : S \times A \times S$. Starting from the initial state, the transitions of the automaton are triggered by the actions that programs fire. If the execution leads to a final, faulty state of the automaton the target is stopped.

The class of properties that can be specified and enforced through these automata coincides with the class of *safety properties*. They play a central role in program monitoring. Indeed, each safety property defines unsafe configurations/states of a system and states that the executions must never reach it. Considering that the instances of safety properties cover many problems of interest, *e.g.*, *deadlock freedom*, the importance of monitoring them is straightforward.

Edit automata have been proposed by Ligatti et al. [6]. Their work mainly focuses on classifying the security monitor according to their capabilities with respect to the executing target. In doing that, they abstract from any property specification formalism. In that way, they identify four different typologies of security automata: *truncation automata*, *suppression automata*, *insertion automata* and *edit automata*. Briefly, a truncation automaton can only halt, *i.e.*, truncate, the execution of its target. Also note that Schneider's automata are a proper subset of this class. Suppression automata are allowed to remove one or more actions from the sequence produced by the program they are responsible to look at. Symmetrically, insertion automata can put new actions inside the received flow. Finally, edit automata capabilities amount to the sum of suppression and insertion. Along with these considerations, the authors name *edit properties* the class of properties that can be enforced by their automata.

Together with the safety properties we have the class of *liveness properties*. Roughly, liveness properties state that a desirable configuration of the system must be always reachable. It is known (*e.g.*, see [7]) that every security property can be expressed as the composition of a safety and a liveness property. Hence, edit automata can enforce a class of properties that contains the safety ones and also part of the liveness properties. This represent an important category of properties that can be effectively enforced. Nevertheless, several liveness properties are out of the scope of run-time monitoring. Mainly, this is due to the finiteness of monitoring. Indeed, at each time instant the monitor is only aware of a finite prefix of the whole execution history of its target.

3 Trust Management

In the previous section we have seen several security solutions. Despite the formal guarantees that they can provide, many of them require strong assumptions to be satisfied, *e.g.*, in terms of actions visibility. Indeed, in many real-life system the evaluation of these assumptions is not straightforward and some uncertainty on the behaviour of the involved components exists. As consequence, some safe execution are discarded because there is not way to formally proof that are secure.

Trust management mechanism can be seen as a way to integrate standard security mechanisms in order to cope with these limitations. Indeed, the lack of information is replaced by trust notions. These kind of approaches are first introduced by Rasmussen & Jonson [8] as *Hard security* for traditional solutions and *Soft security* for solutions that are based on social interactions, of which trust and reputation are considered as the pillars. However, to the best of our knowledge, there is not much work about the integration of trust management and policy enforcement for mobile application in the literature.

The main advantage of the Trust paradigm is the fact that it creates/provides a social capital [9], which means, *"the ability of people to work together for common purposes in groups and organizations* [10].

As illustrated in Fig. 1, on the one hand, the more we trust, the greater is the benefit that can be taken from the social capital. At the same time the risk of being exposed increases. On the other hand, bolstering security has the effect of reducing the risk, but also reduces the benefit through accessibility limitations.

Thus, the aim of extending the Security with Trust (*i.e.*, soft security) is to optimize the benefit/risk ratio, so that increasing the benefit, as well as, decreasing the risk.

In order to enhance security with trust, we have to understand how trust is established and assessed.

Substantial research has been done on the concept of trust in the field of social sciences. The obtained results have been applied in various areas including economics, finance, management, government, and psychology. In recent years, trust has generated considerable interest in the computer science community as the basis of security solutions for various distributed systems, such as *ad-hoc* networks, pervasive environment, Grid, Web services.

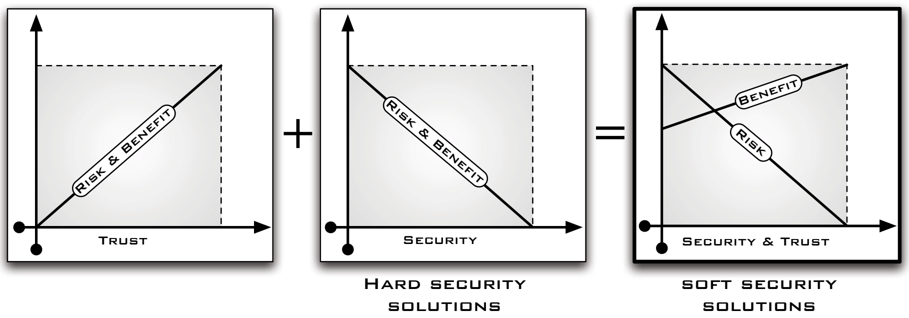


Fig. 1. Risk vs. Benefit

Hence, as people getting increasingly connected virtually as trust management is becoming a central element of today's open distributed digital environment. This is indeed leading to the introduction of various trust management systems and associated trust models, which are customized according to their target applications.

According to [11], *A trustor trusts a trustee with regard to its ability to perform a specific action or to provide a specific service*. Hence, any trust model may basically be defined in terms of the three following elements:

1. *Trustor and Trustee* abstract the representative behaviours of stakeholders from the standpoint of trust management.
2. *Trust relations* serve specifying trust relationships holding among stakeholders, and
3. *Trust assessment* defines how to compute the trustworthiness of stakeholders.

A trust model can be decomposed into (i) the definitions of trust relations (Section 3.1), (ii) trust assessment (Section 5.1), (iii) trust bootstrapping, (iv) and risk management, which plays a key role.

3.1 Trust Relation

Manifestations of trust are easy to recognize because we are confronted to this paradigm everyday, but, at the same time, trust is more complex than it seems considering it manifests itself in many different forms. We identify two types of trust relationships, *i.e.*, *direct* and *indirect*, depending on the number of stakeholders that are involved into the trust relationship (Fig. 2).

Direct trust. A direct trust relationship represents a trust assertion of a subject (*i.e.*, trustor) about another subject (*i.e.*, trustee). It is thus a *one-to-one trust relation* (denoted *1:1*) since it defines a direct link from *1* trustor to *1* trustee. One-to-one trust relations are maintained locally by trustors and represent the trustors' personal opinion regarding their trustees [12]. For example, a one-to-one relation may represent a belonging relationship (*e.g.*, employees trust their company), a social relationship (*e.g.*, trust among friends), or a profit-driven relationship (*e.g.*, a person trusts a trader for managing its portfolio).

Indirect trust. As opposed to a direct trust relationship, an indirect trust relationship represents a subject's trustworthiness based on a third party's recommendation(s). This can be either (i) transitive-based or (ii) reputation-based.

– *Transitive-based trust relations* are *one-to-many* (denoted *1:N*). Such a relation enables *1* trustor (*e.g.*, Alice in Fig. 2(B)) to indirectly assess the trustworthiness of an unknown trustee (*e.g.*, Bob in Fig. 2(B)) through the *recommendations* of a group of trustees (*N*). Hence, the computation of 1:N relations results from the concatenation and/or aggregation of many 1:1 trust relations (arrow T in Fig. 2). The concatenation of 1:1 trust relations usually represents a transitive trust path, where each entity can trust unknown entities based on the recommendation of its trustees. Thus, this relationship is built by composing personal trust relations [13,14]. Furthermore,

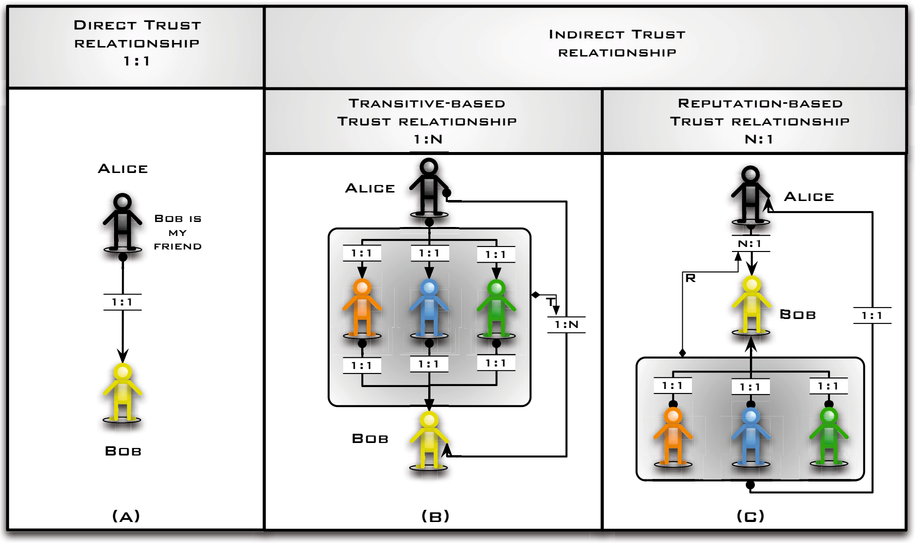

Direct Trust relationship 1:1
Indirect Trust relationship
Transitive-based trust relationship 1:N
Reputation-based trust relationship N:1
Alice
Bob is my friend
1:1
Bob
(A)
Alice
1:1
1:1
1:1
1:N
1:1
1:1
1:1
Bob
(B)
Alice
N:1
R
Bob
1:1
1:1
1:1
1:1
(C)

Fig. 2. Trust Relations

if several trust paths that link the trustor to the recommended trustee exist, the aggregation can be used to aggregate all given trust recommendations [15]. More recently, this kind of relation gained importance by the emergence of WS composition to assess the trustworthiness of Web Services (WS) composition [16,17,18]. In this case, the aggregation of sub-services recommendation allows defining the trust recommendation for the whole composition.

– *Reputation-based trust relations* are *many-to-one* (denoted *N:1*). These relations result from the aggregation of many personal trust relationships of recommenders having the same trustee (see arrow R in Fig. 2). In other words, the a group of recommenders (N) trust a specific subject (*e.g.*, Alice in Fig. 2(C)) to take their personal opinions and to maintain and provide the $reputation$ of trustees (*e.g.*, Bob in Fig. 2(C)). Such reputation can be used, by any entity as a reference, to define its trust relation with unknown ones (*e.g.*, *"I trust you because of your good reputation"*,*"I distrust you because of your bad reputation"*, *"I trust you less than before due to your current bad reputation"* or *"I trust you more than before due to your current good reputation"*).

In the literature, reputation systems are divided into two categories depending on whether they are (i) *Centralized* or (ii) *Distributed*. In the former case, the reputation of each participant is collected and made publicly available at a centralized server (*e.g.*, eBay, Amazon, Google, [19]). In the latter one, reputation is spread throughout the network and each networked entity is responsible to manage the reputation of other entities (*e.g.*, [15,20,21,22]).

3.2 Trust Assessment

Trust assessing, *i.e.*, assigning values to trust relationships, relies on the definition of: (i) trust metrics characterizing how trust is measured, (ii) operations for assessing and composing trust values and (iii) operations for assessing and providing trust values.

Trust metrics. Different metrics have been defined to measure trust. This is due to the fact that one trust metric may be more or less suitable to a certain context. Thus, there is no widely recognized way to assign trust values. Some systems assume only binary values. In [23], trust is quantified by qualitative labels (*e.g.*, high trust, low trust). Other solutions represent trust by a numerical range. For instance, this range can be defined by the interval [-1..1] (*e.g.*, [24]), [0..n] (*e.g.*, [25,13,14]) or [0..1] (*e.g.*, [15,26]. A trust value can also be described in many dimensions, such as (Belief, Disbelief, Uncertainty) [15].

In addition, several definitions exist about the semantics of trust metrics. This is for instance illustrated by the meaning of zero and negative values. For example, zero may indicate lack of trust (but not distrust), lack of information, or deep distrust. Negative values, if allowed, usually indicate distrust, but there is a doubt whether distrust is simply trust with a negative sign, or a phenomenon of its own.

Assessment operations. We define the following main operations for the computation of trust values associated with the trust relations given in Section 3.1 (Table 1): *Updating*, *aggregation*, and *concatenation*.

Table 1. Trust assessment operations

	Setting		Aggregation	Concatenation
	Bootstrapping	Updating		
One-to-One (1:1)	X	X		
One-to-Many (1:N)		X	X	X
Many-to-One (N:1)	X	X	X	

The *setting* operations are mainly performed by trustors to *bootstrap* (i.e., initialize) 1:1 and N:1 trust relationships or to *update* these relationships after receiving personal feedback or third parties recommendation.

The *bootstrapping* operation initializes the *a priori* values of 1:1 and N:1 trust relations. Trust bootstrapping consists of deciding how to initialize trust relations in order to efficiently start the system and also allow newcomers to join the running system [27]. Most existing solutions simply initialize trust relation with a fixed value (*e.g.*, 0.5 [28], a uniform Beta probabilistic distribution [29], etc.). Other approaches include among others: initializing existing trust relations according to given peers recommendations [30]; applying a sorting mechanism instead of assigning fixed values [14]; and assessing trustees into different contexts (*e.g.*, fixing a car, babysitting) and then inferring unknown trust values from known ones of similar or correlate contexts [27,31].

All the solutions dealing with 1:N trust assessment mainly define the *concatenation* and the *aggregation* operations, in order to concatenate and to aggregate trust recommendations by computing the average [14], the minimum or the product [13] of all the intermediary trust values. In the case of Web service composition, some approaches (*e.g.*, [17]) evaluate the recommendation for each service by evaluating its provider, whereas other approaches (*e.g.*, [18]) evaluate the service itself in terms of its previous invocations, performance, reliability, etc. Then, trust is composed and/or aggregated according to the service composition flow (sequence, concurrent, conditional and loop).

Aggregation operations such as Bayesian probability (*e.g.*, [32]) are often used for the assessment of N:1 (reputation-based) trust relations. Trust values are then represented by a beta Probability Density Function [29], which takes binary ratings as inputs (*i.e.*, positive or negative) from all trustors. Thus, the reputation score is refreshed from the previous reputation score and the new rating [19]. The advantage of Bayesian systems is that they provide a theoretically sound basis for computing reputation scores and can also be used to predict future behaviour. Other solutions [22,33] use the fuzzy logic approach, which offers the ability to handle uncertainty and imprecision effectively, and is therefore ideally suited for interpreting trust. In contrast to Bayesian inference, the Fuzzy inference copes with fuzzy inputs, such as assessments of quality, and allows inference rules to be specified using imprecise linguistic terms, such as ”very high quality” or ”slightly late”.

4 Overview of the Security-by-Contract with Trust

In this section we describe the *Security-by-Contract-with-Trust* paradigm as mechanism for enforcing security policies. In order to better explain the reason because we develop this framework, we start by recalling the original idea of *contract-based security* that has lead to the development of the *Security-by-Contract* paradigm described in [34].

4.1 Security-by-Contract Paradigm

The *Security-by-Contract* (S×C) paradigm has been developed, in its first version, for guaranteeing security at run-time on a mobile device. Indeed the idea is to make secure a mobile device while it is executing an applications downloaded from an unknown, and possible malicious, provider.

S×C provides a full characterisation of the contract-based interaction. As a matter of fact, the basic concept of this paradigm is the *contract*.

Definition 1. *A* contract *is a formal complete and correct specification of the behaviour of an application for what concerns security relevant actions (Virtual Machine API call, Operating System Calls). It defines a subset of the traces of all possible security actions.*

Loosely speaking, the contract is an over-approximation of all possible execution behaviours. Every application must be coupled with a corresponding contract, *e.g.*, released by the application developer. Example of contract are [35]:

- The application does not send MMS messages.
- The application only sends messages to determined numbers.
- The application sends only text (or binary) messages.

The other cornerstone of the S×C approach is the concept of *policy*.

Definition 2. *A* policy *is a formal complete specification of the acceptable behaviour of an application to be executed on the platform for what concerns relevant security actions (Virtual Machine API call, Operating System Calls). It defines a subset of the traces of all possible security actions.*

Roughly, it defines a subset of the traces of all possible security actions. Example of policy are [35]:

- The application only receives SMS messages on a specific port.
- The application does not use Bluetooth or IrDA connections.
- The application does not use local socket connections (like 127.0.0.1 or localhost).

Actually, the S×C paradigm works as follows: let us assume that both contracts and policies are specified through the same formalism [36,37]. The code released by a provider is annotated with a contract. When a user receives an application verifies whether the code and the contract actually match by an *evidence checking* procedure. If the check fails then the user can decide to delete the application or to enforce a security policy on it. Otherwise, the system can proceed to verify whether the contract (correctly representing the application) satisfies the user's policy. If this step fails, *i.e.*, the contract does not adhere to the policy, the solution consists in enforcing the active policy on the execution. Finally, if the previous checks were positively passed, the application can be executed with no active runtime monitor.

Looking at the architecture of the S×C framework graphically described in Fig. 3, it is possible to note that the workflow depends on the answers of the two function: "check evidence" and "contract-policy matching".

The check evidence procedure consists of a digital signature of a formal proof. The contract-policy matching function ensures that any security relevant behaviour allowed by the contract is also allowed by the policy. This matching function allows the user that is going to execute the application to understand if the behaviour of the application itself is compliant with the set of policies he has on his device or not without running it. This matching could be done with respect to different behavioural relation, *e.g.*, language inclusion [38] or behavioural relations [39].

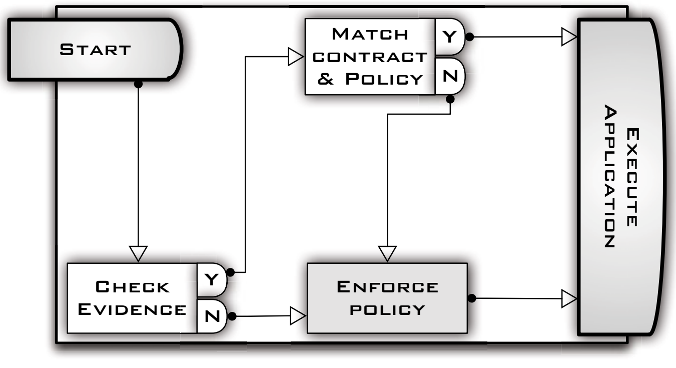


Fig. 3. The Security-by-Contract Workflow

Indeed, the basic idea underlying contract-policy matching is that any sequence of security relevant actions allowed by the contract is also allowed by the policy. Referring to behavioural relation, trace inclusion is thus a suitable candidate for the contract-policy matching. However, for generic contract and policy, the complexity is PSPACE complete, even for finite state systems. We chose to use the simulation relation [40] as a formal compliance relation between contracts and policies. This relation can be efficiently checked on finite state systems. Furthermore, there exists a semi-decision procedure for checking similarity between symbolic transition systems.

In particular, the simulation-based contract-policy matching proposed in [40] consists in checking if a contract transition system is simulated by a policy one, *i.e.*, if, for each transition corresponding to a certain security action of the contract (and so possibly performed by the program), the policy has a similar transition and resulting contract system is yet simulated by the resulting policy one. When the algorithm reports that the contract is simulated by the policy then we can conclude that the contract matches the policy and so the program can be safely executed on the device.

According to the answer of these two functions, the application behaviour is enforced by exploiting the *monitoring/enforcement* infrastructure in order to assure that the security policy set on the device is respected. The enforcing approach has been shown to be feasible on mobile devices. In particular two techniques have been detailed in the literature and exploited for experiments and tools: JVM customization [36] and bytecode in-lining [37]. Briefly, the first replace the standard JVM with a modified one dispatching signals to the monitoring agent whenever a program makes a call to (a subset of) the system APIs. The second mechanism instruments the sequence of bytecode instructions with invocations to the security policy monitor making the program send security signals at run-time. Both approaches use an external component, namely a *Policy Decision Point* (PDP), holding the set of rules that compose the security policy. Moreover the PDP reads the current device state (battery consumption, link strength, available credit) through dedicated internal components. When the PDP receives a request for an action violating the security policy, it answers denying the necessary permission. Then, the system reacts by throwing an exception.

Fig. 4 depicts the two mechanisms described above. Note that the complexity residing in the customised JVM in the first approach is moved to specific security APIs that are attached to the application in the second one.

4.2 Security-by-Contract-with-Trust Paradigm

The Security-by-Contract-with-Trust paradigm, $S{\times}C{\times}T$ for short, has been introduced in [41,42] as a unique framework for managing both security and trust at application execution time. One of the main differences between the Security-by-Contract paradigm and the Security-by-Contract-with-Trust approach is that also the *code* of the application plays a central role in the $S{\times}C{\times}T$ workflow, depicted in Fig. 5. Indeed, the function “check-evidence” is replaced by a trust module that according to the adherence of the application to its contract updates the level of trust of the application itself. It is interesting to note that updating the level of trust of the application implies updating the level of trust of the provider of the application since the device receives the couple application-contract from the provider.

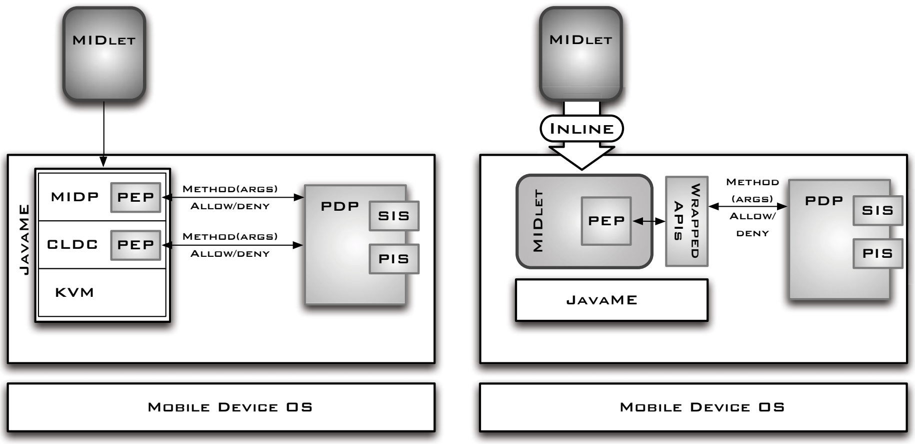


Fig. 4. Enforcement environments using customised JVM (left) and in-lining (right)

The basic idea is the following one: let us consider to have a device and let us suppose to run on it an application, developed by possible unknown developers. As in the S×C paradigm, we assume that contracts and policies are specified through the same formalism. According to Fig. 5, the code is downloaded with its contract. The level of trust is checked, this means that we measure the level of trust that the code adheres to its contract. If the check fails, the code is considered untrusted, so on one side the policy is enforced in order to guarantee security issues, on the other side the contract is monitored in order to log the contract violations. If the monitored execution does not violate its contract the level of trust is upgraded, otherwise it is downgraded. Otherwise, if the code is trusted, the compliance between the contract and the policy is checked in order to understand if the application can be executed without any enforceable mechanisms running on it.

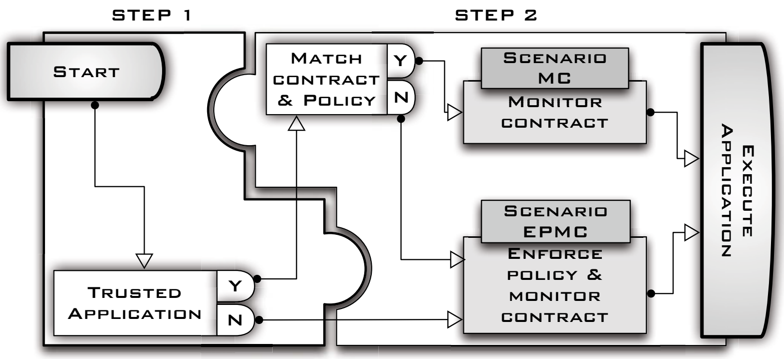
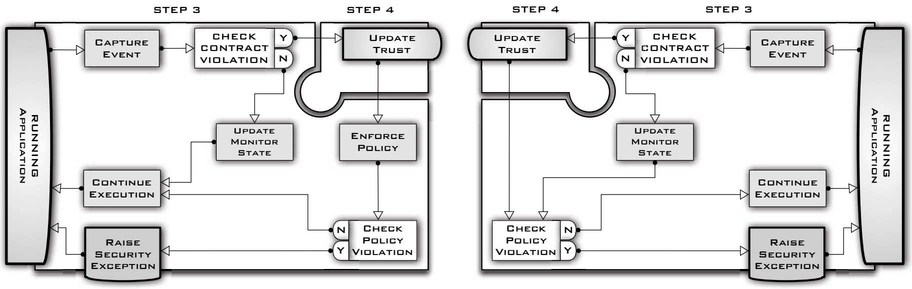


Fig. 5. The Security-by-Contract Workflow



(a) Scenario MC **(b)** Scenario EPMC

Fig. 6. The contract monitoring configurations

Going more in detail, the application lifecycle consists in the following steps:

- **Step 1-*Trust Assessment*:** Each downloaded mobile application comes with a given recommendation rate, which allows the trust module of the user device to decide if the application can be considered as trusted or not (Section 5.1).
- **Step 2-*Contract Driven Deployment*:** According to the trust measure, the security module decides if just monitoring the contract or both enforce the policy and monitoring the contract going into one on the scenarios described in Step 3.
- **Step 3-*Contract Monitoring vs Policy Enforcement Scenarios***: Depending on the chosen scenario the security module is in charge to monitor either the policy or the contract and save the execution traces (logs). Indeed, we have two cases:

  Scenario MC. The monitoring/enforcement infrastructure is required to monitor only the application contract. Indeed, under these conditions, contract adherence also implies policy compliance. If no violation is detected then the application worked as expected. Otherwise, we discovered that a trusted party provided us with a fake contract. More in detail, the contract monitoring works according to the following strategy depicted in Fig. 6a: the contract monitoring receives event signals from the executing code. The execution trace is kept in memory. When a signal arrives, its consistency with respect to the monitored contract is checked. If the contract is respected then its internal monitoring state is updated and the operation is allowed, and a good behaviour is logged (*i.e.*, contract respected). Otherwise, if a violation attempt happens, a security error occurs, and a bad feedback is trigged (*i.e.*, contract violation), and the system switches from contract monitoring to policy enforcement configuration in order to guarantee that the security policy is satisfied. Since an instance of the policy is always present, this operation does not imply a serious computational overhead.

  Scenario EPMC. Since the contract declares some potentially undesired behaviour, policy enforcement is turned on. Similarly to a pure enforcement framework, our system guarantees that executions are policy-compliant. However, monitoring contract during these executions can provide a useful feedback for better tuning the trust vector. Hence, in this scenario, both the policy enforcement and the contract monitoring are active. Indeed, the contract monitoring receives event signals from the executing code and keeps trace of the execution trace. When a signal arrives, its consistency with respect to the monitored contract is checked. If the contract is respected then its internal monitoring state is updated and the operation is allowed, and a good behaviour is logged (*i.e.*, contract respected). Otherwise, if a violation attempt happens, a security error occurs and a violation feedback is logged for the trust module. The policy enforcer is only in charge to following the execution of the application and whenever it attempts to violate the security policy of the device the enforcement mechanism halts the execution in such a way the security policy is satisfied. This configuration is activated on a statistical base (Fig. 6b).

Let us notice that, in both the previous scenarios, contract monitoring plays a central role. Indeed, a contract violation denotes that a trusted provider released a fake contract.

- **Step 4-*Trust Feedback Inference*:** Finally, the trust module parses the S×C×T produced logs and infers trust feedback (Section 5.3).

5 An Application of the S×C×T Framework

In order to better explain the framework, let us consider a Mobile Applications Marketplaces (MAMp), (*e.g.*, Apple AppStore, Cydia, Android Market) in which mobile applications (MAs) are released by a provider (MAP) and customers can download and install

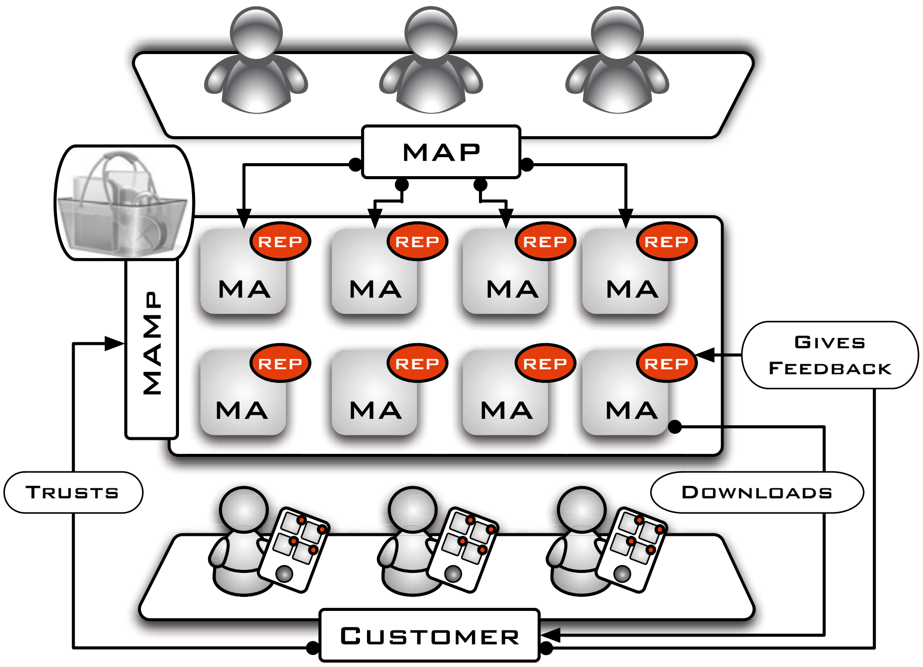


Fig. 7. Mobile Application Marketplace -use case-

them on their devices. Let us also suppose that a trust reputation is associated to each MAP. Reputations are managed by MAMp and updated according to customers' feedback.

Generally speaking, we consider a customer, owner of a mobile device, who needs to add a certain functionality to his own device. The customer asks for an application for doing that. Such MA is downloaded from the MAMp.

The customer decides to download or not the mobile application according to the given recommendation and the trust threshold he has set on his device. Each mobile application comes with its contract released by the provider of the application itself.

Let us suppose that a security policy is embedded on the customer's device. It can be set by the customer himself or by the manufacturer of the device. If the application is downloaded, whenever it is executed, some security mechanisms are needed for guaranteeing that the device's security policy is satisfied.

In this scenario we have applied the Security-by-Contract-with-Trust in order to guarantee that a downloaded application is executed without violating the security policy required by the customer.

Hereafter, we detail the MA lifecycle according to the S×C×T workflow.

5.1 Trust Assessment - Step 1

MAMp computes/maintains trust reputations of mobile applications by aggregating subjective customers' opinions (*i.e.*, 1:1 subjective feedback -Fig. 8-). However, this aggregation process is not sufficient, we have to consider objective feedback such as those provided by monitoring security and privacy. Thus, we introduce a new parameter that reflects the global reputation of a Mobile Application Provider (MAP) in term of

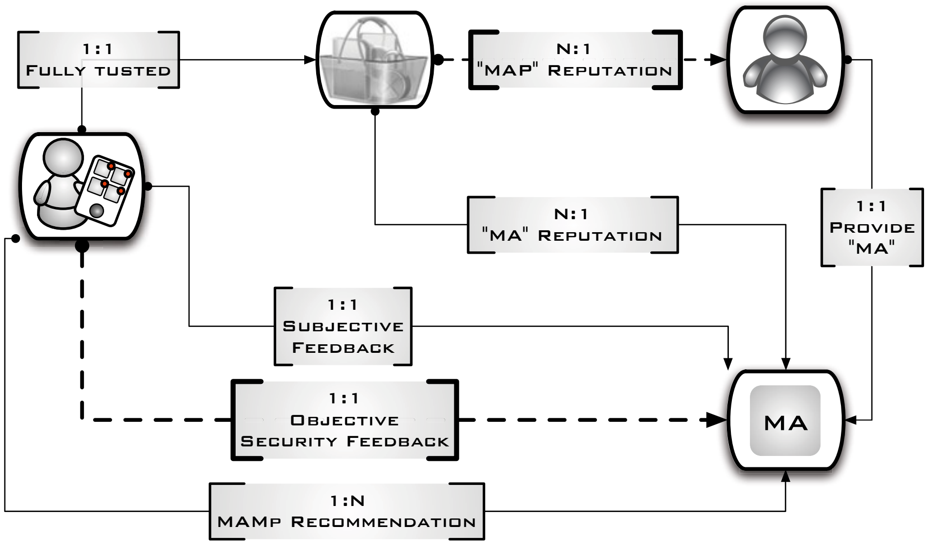


Fig. 8. Mobile Application Marketplace -trust relations-

security behaviour (*i.e.*, N:1 MAP reputation -Fig. 8-) in such a way that the provider's reputation grows according to the number of times that its provided applications satisfies their contract, and vice versa.

In the literature, managing reputation is widely investigated. For our use case, since MAMp are fully trusted, we trust each MAMp to work as a centralised reputation manager that maintains MAs and MAPs reputations. Note that, the MAMp considers only accredited feedback, *i.e.*, all the feedback that can not be reproduced.

Therefore, each customer is able to assess transitively (*i.e.*, 1:N MAMp recommendation -Fig. 8-) through "MAMp" a given recommendation about a MA, which results from the aggregation of subjective and objective feedback as follows:

$$Rec(MAMp, MA) = Rep(MAP) * Rep(MA) \tag{1}$$

Where, the amount of the recommendation is denoted by Rec(MAMp, MA); the $Rep(MA)$ represents the aggregation process of users' subjective opinions that is performed by the MAMp (this value is normalised into the interval [0,1]); the Rep(MAP) falls into the interval [0,1] and denotes the reputation of the provider (MAP) that releases MA. The reputation of MAP is updated according to the S×C×T monitoring.

In order to decide if a given mobile application from MAMp is trusted or not, each customer has to fix its trust threshold Th^0 (bounded between 0 and 1). Hence, the ones that their recommendation is over Th^0 are considered as *trusted*, where those with a recommendation below than Th^0 are *untrusted*.

The trust threshold is set according to the criticality of the mobile application. For instance, a mobile application that manipulates users' localisation can be considered as a risky application and hence will lead to define a high Th^0 close to '1'. Whereas, for a harmless application, Th^0 will be close to '0'.

For bootstrapping, we assume that each MAP is initialised with a low reputation (*i.e.*, 0.1) in order to reduce Whitewashing phenomena [43], where malicious consumers with low reputation leaves the system and then backs with a new identity. Furthermore, neither this kind of attack nor the Sybil attack [43] are efficient, since in MAMp is very tedious to create multiple accounts especially if it requires an expense from the part of the attacker.

5.2 Monitoring/Enforcement Process - Step 2 and Step 3

Once the Trust module established the threshold according to the application criticality and once it gets the trust recommendation of the MA, we have the following two possibilities:

(i) The customer does not trust that the MA behaviour adheres to the MA contract. In this case the contract policy matching is not performed since it does not provide any guarantee about the compliance between the MA behaviour and the policy. Hence we turn to the *Enforce policy & Monitor Contract* scenario (EPMC scenario of Fig. 3). This allows us to guarantee that the security policy is respected by applying the enforcement mechanism and trust feedback are provided according to the contract monitoring answer.

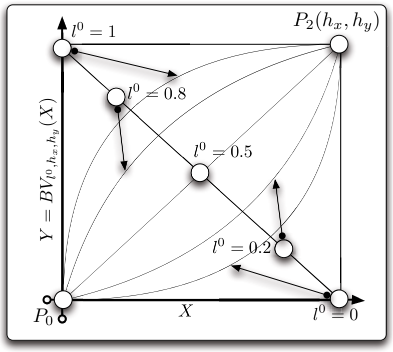
(ii) The customer trusts the MA, *i.e.*, the customer trusts that the MA behaviour adheres to the MA contract. In this case we check if the contract satisfies the policy by the *Contract-Policy Matching* function. Two subcases arise:

– The contract satisfies the policy. This allows the customer to establish that the MA satisfies also the security policy. For that reason, the MA is executed and we turn into the *Monitor Contract* scenario (MC scenario of Fig. 3).
– The contract does not satisfy the policy. Hence, we are not able to infer anything about the compliance between the MA behaviour and the security policy. The enforcement mechanism is applied for guaranteeing that the security policy is respected. Thus, also in this case, we turn into the *Enforce Policy & Monitor Contract* scenario (EPMC scenario of Fig. 3). The contract monitoring can provide a useful feedback for better tuning MAPs' reputation. The contract monitoring is performed only for providing feedback to the trust module. It is performed on statistical bases according to a probability called P_{mon}. P_{mon} is computed according to the level of the given recommendation and the remaining battery life. Indeed, P_{mon} is inversely proportional to the measure of trust of the application and proportional to the remaining battery life. For instance, the more trusted the application is or the lower the battery life is, the less frequent the contract monitoring is performed.

We implement the behaviour function BV [14] in order to compute P_{mon}. The *behaviour function* BV is monotonically increasing function, and is able to oscillate from parabolic to hyperbolic shape according to a behaviour level called l^0.

Note that it is possible to use other mathematical functions (logarithmic, exponential or stepwave) to assess the monitoring probability but the originality of using Bezier curves comes from the easiness of plotting different monotonic and increasing curves (*i.e.*, logarithmic-like and exponential-like) with a unique function taking as inputs only two parameters, namely the curve thresholds and the degree of the curvature.

Definition 3. *[14] [BV_{l^0,h_x,h_y}: The behaviour Function] The BV function is the Cartesian form of a quadratic Bezier curve. Thus, the BV function is able to draw smooth curves (see Figure 9) which are achieved through three points P_0, P_1 and P_2, starting at P_0 going towards P_1 and terminating at P_2, where:*
- $P_0(0,0)$ *is the origin point.*
- $P_2(h_x,h_y)$ *is called the threshold point, where the h_x and h_y represent respectively the abscissa and the ordinate thresholds.*



$$\begin{aligned} BV : [0,h_x] &\longrightarrow [0,h_y] \\ X &\longrightarrow Y \end{aligned}$$

$$BV_{l^0,h_x,h_y}(X) =$$

$$\begin{cases} \frac{(h_y-2b_y)}{4b_x^2}X^2 + \frac{b_y}{b_x}X & si\ (h_x - 2b_x = 0) \\ (h_y - 2b_y)(\propto(X))^2 + 2b_y \propto(X), & si\ (h_x - 2b_x \neq 0) \end{cases}$$

$$Where \begin{cases} \propto(X) = \frac{-b_x + \sqrt{{b_x}^2 - 2b_x * X + h_x * X}}{h_x - 2b_x} \\ 0 \le b_x \le h_x \wedge h_x > 0 \\ b_x = (1 - l^0) \cdot h_x \wedge b_y = h_y \cdot l^0 \end{cases}$$

Fig. 9. The BV function

- $P_1(b_x, b_y)$ *defines the behaviour point. The coordinate of this point* (b_x, b_y) *are guided by a given level '*l^0*' (*$l^0 \in [0,1]$*) through the diagonal of the rectangle that is defined by* P_0 *and* P_2*,.*

Thus, by aligning the behaviour level l^0 with the trust level of the application $Rec(MAMp, MA)$ (*i.e.*, $l^0 = 1 - Rec(MAMp, MA)$) and by representing, through the abscissa, the battery life $Battery_{remain}$, we obtain the following equation:

$$\mathcal{P}_{mon}(C) = BV_{(1-Rec(MAMp,MA)),1,1}(Battery_{remain}) \tag{2}$$

Table 2 shows some examples of monitoring probabilities that are obtained by varying the trust recommendation value of the mobile application and the percentage of the remaining battery. Hence, as expected, the probability decreases when the battery life falls (proportionately) or the given recommendation become higher (inversely proportional).

Table 2. Monitoring probability examples ($P_{mon}(C)$)

$Rec(MAMp, MA)$ \ $Battery_{remain}$	20%	40%	60%	80%	100%
0.2	0.47	0.70	0.84	0.93	1.00
0.5	0.20	0.40	0.60	0.80	1.00
0.8	0.06	0.16	0.30	0.53	1.00
1.0	0.01	0.05	0.16	0.31	1.00

5.3 Trust Feedbacks -Step 4

According to the previous steps, a trust log that reflects the behaviour of the deployed application throughout the monitoring process is generated. Table 2 represents all possible logs by crossing all possible behaviours with a trusted and an untrusted case. Thus, four types of feedback are highlighted, namely, Highly Reward, Reward, Penalise, and Highly Penalise.

As illustrated in Table 2, for a trusted application, it is obvious to reward the corresponding MAP if that application respects its contract but we have to turn on the EPMC scenario, and to highly reward if the application matches the contract and we are in the MC scenario. On the contrary, we highly penalise MAP if its recommended application violates its contract.

Table 3. Feedback types

Trust \ Security	Respect the contract		Violate the contract	
	MC scenario	EPMC scenario	MC scenario	EPMC scenario
Trusted	Highly reward	Reward	Highly penalise	
Untrusted	——	Reward	——	Penalise

However, for an untrusted application, the Security-by-Contract is only applied under the EPMC scenario, and it check the contract and enforce the policy. In this case, we adopt a reserved behaviour, by just rewarding if the contract is respected and penalising otherwise.

In order to reduce the impact of malicious customers on our framework (*i.e.*, Slandering and denial of services attacks [43]), in which they try to send multiple bad feedback for good application or several good feedback for bad ones, each customer can only provide one reward and one penalty feedback for each mobile application. Furthermore, as mentioned above, each given recommendation is processed, by our model, proportionately to the reputation of its sender.

Then, MAMp performs the Algorithm 1, with the given trust feedback, to update MAP's reputation which directly results in updating all the recommendation of the applications of that provider.

MAMp updates the MAP's reputation (Algorithm 1 line 18) by increasing or decreasing current evaluation proportionately to α and also to the quantitative variable $Feedback\_amount$. The $Feedback\_amount$ is respectively equal to '0.5' for reward or penalise and '1' for highly reward or highly penalise.

The parameter α ($\alpha \in [a, b]$) defines how significant is the impact of new actions will be on the previous acquired trust history. We choose to bound α into the interval [a, b], where a is close to 0 (*e.g.*, $a = 0.2$) and b is close to '1' (*e.g.*, $a = 0.8$). Thus, $\alpha = a$ means that past acquired history is highly relevant, whereas $\alpha = b$ gives the highest impact to new actions.

Algorithm 1 . Update trust

```
Require:
    $MA$: The monitored Mobile Application
    $Feedback \in \{"Reward", "Highly\ reward", "Penalise", "Highly\ penalise"\}$
    $\alpha \in [a, b]$

1.  $MAP = Pr(MA)$ $\{Pr(MA)$ returns the MAP of MA$\}$
2.  $OldPr = Rep(MAP)$ $\{Rep(MAP)$ returns or updates the current reputation of MAP$\}$
3.  if $Feedback = "Reward"$ or $Feedback = "Highly\ reward"$ then
4.    if $Feedback = "Reward"$ then
5.      Feedback_amount=0.5
6.    else if $Feedback = "Highly\ reward"$ then
7.      Feedback_amount=1
8.    end if
9.    $NewPr = OldPr + Feedback\_amount * (1 - OldPr)$
10. else if $Feedback = "Penalise"$ or $Feedback = "Highly\ penalise"$ then
11.   if $Feedback = "Penalise"$ then
12.     Feedback_amount=0.5
13.   else if $Feedback = "Highlypenalise"$ then
14.     Feedback_amount=1
15.   end if
16.   $NewPr = OldPr - Feedback\_amount * OldPr$
17. end if
18. $Rep(MAP) = (1 - \alpha) * OldPr + \alpha * NewPr$
```

In order to automatically infer the value of α, two statements are considered:

- α is proportional to the recommendation value. In fact, if the given recommendation of the application is low and a new good behaviour is monitored, MAMp has to slightly update trust values (*i.e.*, α is close to a). Whereas, if the given recommendation of the application is high and a bad behaviour is trigged, MAMp has to give a high impact to new computed feedback (*i.e.*, α is close to b).
- α is proportional to Th^0 (*i.e.*, application criticality) in case of a penalty and inversely proportional to Th^0 in case of a reward. Indeed, risky applications have to be less rewarded than harmless applications and inversely, risky applications have to be more strongly penalised than harmless applications.

Thus, as for contract monitoring process, α can be inferred using the BV function as follows:

$$\alpha = \begin{cases} BV_{Th^0,1,(b-a)}(REC) + a & if \text{ Feedback} \in \{Penalise, Highly\,penalise\} \\ BV_{(1-Th^0),1,(b-a)}(REC) + a & if \text{ Feedback} \in \{Reward, Highly\,reward\} \end{cases}$$

$$\text{W}here\ REC = Rec(MAMp, MA) \tag{3}$$

Table 4 shows some examples of $alpha$ that are obtained by varying the criticality of the mobile application and the amount of the recommendation.

Table 4. α computing examples

	Penalty				Reward			
Th^0 \ $Rec(MA)$	0.2	0.4	0.6	0.8	0.2	0.4	0.6	0.8
0.8	0.48	0.62	0.70	0.76	0.24	0.30	0.38	0.52
0.2	0.24	0.30	0.38	0.52	0.48	0.62	0.70	0.76

6 Conclusion and Future Work

In this paper we have presented a contract-based and trust-driven run-time enforcement mechanism in which both the notions of security and trust are integrated in a unique approach. Indeed, starting by describing the Security-by-Contract paradigm, in which the notion of trust was not taken into account, we described the Security-by-Contract-with-Trust framework.

The main novelty of the Security-by-Contract-with-Trust (S×C×T) framework with respect to the previous one, consists in the contract monitoring scenario that allows us to manage the trust level of an application. As a matter of fact, at deploy-time, the monitoring structure is decided depending on both the application contract and the credentials (*i.e.*, trust measure) of the contract releaser. Furthermore, at run-time, a trusted program violating its contract leads to a correction of the trust relationship with the provider and activates the policy enforcement configuration of our system.

In order to better explain how the Security-by-Contract-with-Trust framework works, we showed how it can be applied to a Mobile Application Marketplace (MAMp) scenario. In particular, we provided a possible trust management strategy for managing trust feedback according to the concept of mobile application criticality, *i.e.*, mobile application developers are rewarded or penalized according to its level of the recommendation that is provided by the Mobile Application Marketplace and the criticality of the application. The information about the amount of the recommendation can be retrieved from the MAMp and the contract of the application that the developer has to send in addition to the application itself, respectively. This means that we have a unified architecture for managing both security and trust. According to the level of trust and the trust threshold set by the user on the device, the Security-by-Contract-with-Trsut mechanism monitors the contract or both enforces the policy and monitor the contract.

This mechanism allows the owner of the device to be sure to execute downloaded application in a secure way and, at the same time, to be able to take trace of trusted and untrsuted provider for future interactions.

Many future directions are viable. Mainly our trust management strategy is still a work in progress. Currently, trust weights can only decrease monotonically as a consequence of contract violations with the only exception of a direct intervention of the user. Also the contracting infrastructure can be further improved. As a matter of fact, in this work we only referred to single-contract applications. However, we can extend our model in order to accept more contract instances for a single program. This scenario seems to be realistic and open new directions of investigation.

We also envision to extend the trust module to allow banishing malicious applications and/or developers from MAMp by taking into account the time fading aspect.

Finally, similarly to [37], we plan to implement a working prototype for testing the practical feasibility of our approach including performance and resources consumption analysis.

References

1. Jøsang, A., Keser, C., Dimitrakos, T.: An we manage trust?, pp. 93–107 (2005)
2. Necula, G.C.: Proof-carrying code. In: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL 1997), pp. 106–119 (1997)
3. Sekar, R., Venkatakrishnan, V., Basu, S., Bhatkar, S., DuVarney, D.C.: Model-carrying code: a practical approach for safe execution of untrusted applications. In: SOSP 2003: Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, pp. 15–28 (2003)
4. Gong, L.: Java Security: Present and Near Future. IEEE Micro 17(3), 14–19 (1997)
5. Schneider, F.B.: Enforceable security policies. ACM Transactions on Information and System Security 3, 2000 (1998)
6. Ligatti, J., Bauer, L., Walker, D.: Edit automata: Enforcement mechanisms for run-time security policies. International Journal of Information Security 4, 2–16 (2005)
7. Lamport, L.: Proving the correctness of multiprocess programs. IEEE Trans. Softw. Eng. 3, 125–143 (1977)
8. Rasmusson, L., Jansson, S.: Simulated social control for secure Internet commerce. In: Proceedings of the 1996 Workshop on New Security Paradigms, pp. 18–25. ACM, New York (1996)

9. Fukuyama, F.: Trust: The social virtues and the creation of prosperity. Free Press, New York (1996)
10. Coleman, J.: Social capital in the creation of human capital. American Journal of Sociology 94(1), 95–120 (1988)
11. Grandison, T., Sloman, M.: A survey of trust in internet applications. IEEE Communications Surveys & Tutorials 3(4), 2–16 (2009)
12. Kautz, H., Selman, B., Shah, M.: Referral Web: combining social networks and collaborative filtering. Communications of the ACM 40(3), 63–65 (1997)
13. Abdul-Rahman, A., Hailes, S.: A distributed trust model. In: NSPW: New Security Paradigms Workshop, pp. 48–60. ACM Press, New York (1997)
14. Saadi, R., Pierson, J.M., Brunie, L.: Establishing trust beliefs based on a uniform disposition to trust. In: ACM SAC: Trust, Reputation, Evidence and other Collaboration Know-how track. ACM Press, New York (2010)
15. Jøsang, A., Pope, S.: Semantic constraints for trust transitivity. In: APCCM: 2nd Asia-Pacific Conference on Conceptual Modelling, pp. 59–68. Australian Computer Society, Inc., Newcastle (2005)
16. Nepal, S., Malik, Z., Bouguettaya, A.: Reputation Propagation in Composite Services. In: Proceedings of the 2009 IEEE International Conference on Web Services, vol. 00, pp. 295–302. IEEE Computer Society, Los Alamitos (2009)
17. Paradesi, S., Doshi, P., Swaika, S.: Integrating Behavioral Trust in Web Service Compositions. In: Proceedings of the 2009 IEEE International Conference on Web Services, pp. 453–460. IEEE Computer Society, Los Alamitos (2009)
18. Kim, Y., Doh, K.: Trust Type based Semantic Web Services Assessment and Selection. In: Proceedings of ICACT, pp. 2048–2053. IEEE Computer, Los Alamitos (2008)
19. Nurmi, P.: A bayesian framework for online reputation systems. In: International Conference on Internet and Web Applications and Services/Advanced International Conference on Telecommunications, AICT-ICIW 2006, pp. 121–121 (2006)
20. Xiong, L., Liu, L.: A reputation-based trust model for peer-to-peer ecommerce communities. In: 4th ACM Conference on Electronic Commerce, pp. 228–229 (2003)
21. Zhou, R., Hwang, K., Cai, M.: Gossiptrust for fast reputation aggregation in peer-to-peer networks. IEEE Transactions on Knowledge and Data Engineering, 1282–1295 (2008)
22. Song, S., Hwang, K., Zhou, R., Kwok, Y.: Trusted P2P transactions with fuzzy reputation aggregation. IEEE Internet Computing 9(6), 24–34 (2005)
23. Zimmermann, P.R.: The official PGP user's guide. MIT Press, Cambridge (1995)
24. Marsh, S.: Formalising Trust as a Computational Concept. PhD thesis, University of Stirling, Scotland (1994)
25. Golbeck, J., Hendler, J.: Filmtrust: Movie recommendations using trust in web-based social networks. In: CCNC: IEEE Consumer Communications and Networking Conference, Las Vegas, NV, USA, pp. 282–286. IEEE Computer Society, Los Alamitos (2006)
26. Theodorakopoulos, G., Baras, J.S.: Trust evaluation in ad-hoc networks. In: 3rd ACM Workshop on Wireless Security, pp. 1–10. ACM Press, New York (2004)
27. Quercia, D., Hailes, S., Capra, L.: TRULLO-local trust bootstrapping for ubiquitous devices. In: Proc. of IEEE Mobiquitous (2007)
28. Haque, M., Ahamed, S.: An omnipresent formal trust model (FTM) for pervasive computing environment. In: 31st Annual International Computer Software and Applications Conference, COMPSAC 2007, vol. 1 (2007)
29. Jsang, A., Ismail, R.: The beta reputation system. In: Proceedings of the 15th Bled Electronic Commerce Conference, pp. 17–19 (2002)
30. Rahman, A., Hailes, S.: Supporting trust in virtual communities. In: IEEE Hawaii International Conference on System Sciences, p. 6007 (2000)

31. Ahamed, S., Monjur, M., Islam, M.: CCTB: Context correlation for trust bootstrapping in pervasive environment. In: 2008 IET 4th International Conference on Intelligent Environments, pp. 1–8 (2008)
32. Mui, L., Mohtashemi, M., Ang, C., Szolovits, P., Halberstadt, A.: Ratings in distributed systems: A bayesian approach. In: Proceedings of the Workshop on Information Technologies and Systems (WITS), pp. 1–7. Citeseer (2001)
33. Aringhieri, R., Damiani, E., Di Vimercati, S.D.C., Paraboschi, S., Samarati, P.: Fuzzy techniques for trust and reputation management in anonymous peer-to-peer systems: Special topic section on soft approaches to information retrieval and information access on the web. JASIST: Journal of the American Society for Information Science and Technology 57(4), 528–537 (2006)
34. Dragoni, N., Martinelli, F., Massacci, F., Mori, P., Schaefer, C., Walter, T., Vetillard, E.: Security-by-contract (SxC) for software and services of mobile systems. In: At your service - Service-Oriented Computing from an EU Perspective. MIT Press, Cambridge (2008)
35. Labs, T.: Ontology. S3MS CP-2006-RT-503-0.3 (2006)
36. Castrucci, A., Martinelli, F., Mori, P., Roperti, F.: Enhancing java ME security support with resource usage monitoring. In: Chen, L., Ryan, M.D., Wang, G. (eds.) ICICS 2008. LNCS, vol. 5308, pp. 256–266. Springer, Heidelberg (2008)
37. Costa, G., Martinelli, F., Mori, P., Schaefer, C., Walter, T.: Runtime monitoring for next generation java me platform. Computers & Security (2009)
38. Desmet, L., Joosen, W., Massacci, F., Philippaerts, P., Piessens, F., Siahaan, I., Vanoverberghe, D.: Security-by-contract on the .net platform, Oxford, UK, vol. 13, pp. 25–32. Elsevier Advanced Technology Publications, Amsterdam (2008)
39. Milner, R.: Communicating and mobile systems: the π-calculus. Cambridge University Press, Cambridge (1999)
40. Greci, P., Martinelli, F., Matteucci, I.: A framework for contract-policy matching based on symbolic simulations for securing mobile device application. In: ISoLA, pp. 221–236 (2008)
41. Costa, G., Dragoni, N., Lazouski, A., Martinelli, F., Massacci, F., Matteucci, I.: Extending security-by-contract with quantitative trust on mobile devices. In: Proceeding of CISIS 2010, The Fourth International Conference on Complex, Intelligent and Software Intensive Systems, Krakow, Poland, pp. 872–877. IEEE Computer Society, Los Alamitos (2010)
42. Costa, G., Dragoni, N., Issarny, V., Lazouski, A., Martinelli, F., Massacci, F., Matteucci, I., Saadi, R.: Extending security-by-contract with quantitative trust on mobile devices. Journal of Wireless Mobile Networks, Ubiquitous Computing and Dependable Applications (JOWUA) 1(4), 75–91 (2011) ISSN (print): 2093-5374, ISSN (on-line): 2093-5382
43. Hoffman, K., Zage, D., Nita-Rotaru, C.: A survey of attack and defense techniques for reputation systems. ACM Computing Surveys (CSUR) 42(1), 1–31 (2009)

Modeling Spatial and Temporal Variability with the HATS Abstract Behavioral Modeling Language[⋆]

Dave Clarke[1], Nikolay Diakov[2], Reiner Hähnle[3], Einar Broch Johnsen[4], Ina Schaefer[5], Jan Schäfer[6], Rudolf Schlatte[4], and Peter Y.H. Wong[2]

[1] Katholieke Universiteit Leuven, Belgium
[2] Fredhopper B.V., Amsterdam, The Netherlands
[3] Chalmers University of Technology, Sweden
[4] University of Oslo, Norway
[5] Technische Universität Braunschweig, Germany
[6] Technische Universität Kaiserslautern, Germany

Abstract. The *Abstract Behavioral Specification* (ABS) language facilitates to precisely model the behavior of highly configurable, distributed systems. Its basis is Core ABS which is a strongly typed, abstract, object-based, concurrent, fully executable modeling language. Spatial variability of ABS models is represented by feature models, delta modules containing modifications of ABS models, product line configurations linking delta modules with product features and product selections specifying actual product instances. Temporal variability is captured by dynamic delta modules that can be applied to perform runtime updates. The feasibility of ABS is demonstrated by modeling an industrial-scale web merchandising system.

1 Introduction

Contemporary software development faces recurring challenges over many different application domains: software systems are concurrent and distributed, they exhibit a large variety of features and deployment scenarios, their requirements change frequently, and new requirements arise unexpectedly. All of these characteristics are increasingly difficult to address on the level of implementation languages, such as C/C++, C#, or Java. Even when it is possible, the result is often a large gap between the implemented system and the requirements documentation resulting in high validation and maintenance costs and impeding traceability.

Further, major IT-trends under way pose new challenges: a prerequisite for cloud computing is the ability to abstract away from physical resource allocation, load distribution, the architecture of the execution platform, etc. This implies the need to specify intended behavior without referring to concrete resources.

[⋆] This research is funded by the EU project FP7-231620 HATS: Highly Adaptable and Trustworthy Software using Formal Models (`http://www.hats-project.eu`)

Likewise, the emergence of cyber-physical systems and the internet of things emphasize the need for *abstract* behavioral description of highly configurable and diverse systems.

Model-centric approaches to system development are gaining rapidly in popularity in order to provide an abstract representation of system structure and behavior. There is a lot of research involving feature description languages [5], architectural languages for components [17], or UML and state machine-based notations [1,12,25]. Development processes, such as software product line engineering [18] distinguish between generic artifact and product-level system development and are specifically designed to use (and reuse) high-level artifacts. Hence, modeling languages capturing system diversity are required to deal with the variability of generic development artifacts. The main limitation of existing modeling approaches is their insularity and the lack of a unified formal semantics. Both is necessary, however, to provide a future generation of development tools that can, for example, generate code from models that is guaranteed to be sound, generate test cases that have a guaranteed coverage, or ensure that the implementation of features obeys their application constraints.

The HATS (Highly Adaptable and Trustworthy Software using Formal Models) project develops an executable modeling language called *Abstract Behavioral Specification* (ABS) language and an accompanying tool framework that promises to overcome the mentioned shortcomings: it facilitates to model precisely the behaviour of highly configurable, distributed systems in an "end-to-end" manner. This means that not only the (concurrent) implementation of features is captured, but also the feature space and the dependencies among them. Furthermore, ABS includes language concepts to represent model evolution, e.g., due to changing requirements. In addition, it is possible to formally specify properties of systems modeled with ABS in form of behavioral contracts. The ABS modeling language aims to fill the gap between structural high-level modeling languages, such as UML, and implementation-close formalisms, including programming languages.

Fig. 1 shows the different language layers constituting the ABS. At its core, ABS is a state-of-the-art, strongly typed, abstract, concurrent, object-based modeling language (Core ABS) that is fully executable. Shared memory is only permitted among closely collaborating synchronous groups of objects. Otherwise, objects communicate asynchronously and use message passing to update the state. This core language is described in Section 2. While ABS is an object-based language and, hence, compatible with the UML world, code reuse by inheritance, which tends to be brittle, is excluded. Instead, system diversity in ABS is captured by delta modeling [19,20,22,21] which represents a set of systems by a designated core system and a set of system deltas specifying modifications to the core system. Delta modeling is an incremental composition technique that is highly compatible with feature-oriented software development [2] and also a good match for agile and evolutionary development approaches [3].

In Section 3, we describe how *spatial variability* is captured using the different language layers of the ABS (cf. Fig. 1). Spatial variability is concerned with the

Language	Rôle
Core ABS	Specifies core behavioural modules (independent of extensions)
Micro Textual Variability Language (μTVL)	Feature models, attributes and constraints on them
Delta Modelling Language (DML)	Modifications to core behavioural modules
Product Line Configuration Language (CL)	Links features and delta modules, configures deltas with attributes
Product Selection Language (PSL)	Feature and attributes selections plus product initialisation block

Fig. 1. Language definitions in ABS

modeling of anticipated features, as well as known system diversity and deployment scenarios. First, we introduce μTVL (Micro Textual Variability Language) to represent feature models and feature constraints. Second, we describe DML (Delta Modelling Language) that captures variability of Core ABS models by the concepts of delta modeling. Finally, we present CL (Product Line Configuration Language) for configuring a product line of ABS models and PSL (Product Selection Language) for deriving particular products from an ABS product line.

Even more difficult than spatial variability, but of growing importance is *temporal variability* or evolvability in time. The main difference to spatial variability is that temporal variability is not known in advance and cannot be anticipated. In Section 4, we show how delta modeling can be adapted to deal with temporal evolution of ABS product lines including the possibility to perform runtime system updates.

The measure of success for an ambitious and holistic approach such as it is pursued in HATS is whether one can model not only toy examples, but real industrial scenarios. In Section 5, we present an ABS modeling example that is taken from production code used in the distributed web merchandising platform—Fredhopper Access Server [11]. We conducted this industrial-strength case study along with the development of the ABS language and tools, in order to provide valuable early feedback to guarantee strong results. Section 6 describes the ABS tool set consisting of a parser, type checker, editor, debugger, and code generators. We conclude in Section 7 with an evaluation of what has been achieved with HATS ABS so far.

2 Core ABS

Core ABS (or simply ABS in the following) is an object-based modeling language. With its object-based model structure, it fits well with modeling languages used in object-oriented analysis and design, such as UML. However, code reuse via class-based inheritance is excluded. Instead, variability and code reuse in ABS models is achieved by specific language constructs as explained in Section 3.

ABS is designed to model distributed systems that communicate asynchronously by exchanging messages. The concurrency model of ABS is similar to that of JCoBox [24], which generalizes the concurrency model of Creol [15,4] from single concurrent objects to concurrent object groups (COG). COGs can be regarded as object-based runtime components, which have their own heap of objects and solely communicate via asynchronous method calls. The behavior of a COG is represented by cooperative multi-tasking. Cooperative multi-tasking guarantees data-race freedom inside a COG and enables the safe combination of active and reactive behavior.

Beside the object-based part, the core language supports user-defined data types with (non-higher-order) functions and pattern matching. This functional sublanguage of ABS is largely orthogonal to the object-based part and is intended to model data. As such data is immutable, it can safely be exchanged between COGs. Using functional data types to realize most internal data structures of COGs will simplify the specification and verification of COGs.

The ABS language contains non-deterministic constructs; in particular, the outcome of executing concurrency primitives is non-deterministic. While underspecification is used to realize abstraction on data, non-deterministic execution semantics is the prerequisite for abstracting behavior. As ABS is a modeling language, we do not want to make any a priori assumptions about, for example, a concrete scheduling mechanism. Underspecification and non-determinism do not preclude executability: an unknown value is still a value and the outcome of a non-deterministic statement is a set of possible successor states from which one can be picked in simulation and visualization.

In this section, we first describe how to represent data in the ABS. Second, we explain the object-based fragment and the module system of the ABS. Finally, we present the concurrency model that is based on COGs and cooperative multitasking. A complete description of all ABS features is in [10].

2.1 Data Types

Built-In Data Types. ABS does *not* have primitive types, but a number of built-in data types and operators to work with basic values.

The Unit Value. To express that an expression has *no* value, ABS has the data type `Unit`, which has only one, identically named, constructor `Unit`. The `Unit` type is typically used for methods that do not have a return value.

Logical Values. ABS supports logical values by the `Bool` data type. It has the two constructors `True` and `False`. The defined operators on `Bool` are *equality* (`==`), *unequality* (`!=`), *negation* (`~`), *logical and* (`&&`), and *logical or* (`||`).

Example:

```
~((True && False) || True)
```

Numbers. ABS supports unbounded integers by the data type `Int`. Integers are constructed by using *integer literals*, which are positive numbers of an arbitrary length, e.g., `0, 1, 3434, 4711, 42`. ABS supports the standard arithmetic operations on `Int` with the usual precedences, i.e., *negation* (`-`), *addition* (`+`), *subtraction* (`-`), *multiplication* (`*`), *division* (`/`), and *modulo* (`%`).

Example:

```
((-5+6)*4)/(2%1)
```

Character Sequences. Character sequences are represented by the data type `String`. Strings are constructed by using *string literals*, which are sequences of characters enclosed by double quotes (`"`). There is no special data type for single characters, as a single character can be regarded as a string of length 1. The *concatenation* operator (`+`) can be used to concatenate two strings. The length of a string can be obtained by the `length` function.

Example:

```
"Hello" + "World"
```

Algebraic Data Types. Immutable values can be defined in ABS by *algebraic data types*. The possible values of data types are defined by a finite set of *data type constructors*. Constructors can have a finite list of parameters, which can refer to built-in types, algebraic data types, or *reference types* (see Section 2.2). The names of data types and constructors start with an upper case letter. The following example defines the data type `Fruit` with three constructors, and the data type `Juice` with the two constructors `Pure` and `Mixed`. With these data types, we can create a cherry-banana juice.

Example:

```
data Fruit = Apple | Banana | Cherry;
data Juice = Pure(Fruit) | Mixed(Juice, Juice);
Mixed(Pure(Cherry),Pure(Banana))
```

Parametric Data Types. ABS also supports data types with *type parameters* to define generic data types. A typical example is a `List` data type that should be generic in the types of its elements. In ABS, there is the predefined `List` data type, which is defined as follows:

Example:

```
data List<T> = Nil | Cons(T, List<T>);
```

Type Synonyms. To define shortcuts for types, ABS knows *type synonyms*, which are defined by using the **type** keyword. Semantically, a type synonym is equivalent to its aliased type.

Example:

```
type Catalog = Map<String, Product>;
```

Functions. Functions in ABS are used for working with data types. They are always side-effect free. A function is defined by using the **def** keyword. Also, functions can have type parameters to abstract from concrete types. For example, the predefined `head` function is over a parametric data type `A` is declared as follows:

Example:

```
def A head<A>(List<A> list) = ...
```

Pattern Matching. In order to conveniently work with algebraic data types, ABS supports the pattern matching. A *pattern* can be a *bound variable*, in which case it matches against its value, a *free variable*, in which case the variable matches everything and is bound to the matched value, the placeholder _ (underscore) which matches anything, but does not establish a binding, and a *constructor pattern*, in which case the value must match the corresponding constructor. Pattern matching can be defined using the **case** expression. In the following example, pattern matching is used to get the set of all ingredients of a given juice. The data type `Set`, with its constructors `Insert` and `EmptySet`, as well as the function `union` are predefined in ABS.

Example:

```
def Set<Fruit> ingredients(Juice juice) =
  case juice {
    Mixed(j1,j2) => union(ingredients(j1),ingredients(j2));
    Pure(fruit) => Insert(fruit,EmptySet);
  } ;
```

2.2 Object-Based Programming

Classes. ABS models are structured into *classes*. A class declaration consists of the class name, a list of constructor arguments, a list of interfaces that the class implements, a list of fields (instance variables), an init block, and a list of methods. All of these except the class name are optional.

Example:

```
class IPing(Pong pong, Int pingCount) implements Ping {
  Int pingsLeft = pingCount; // A field definition

  // The init block contains non−trivial field initializations.
  // All constructor arguments are fields as well and can be used here.
  {
    ...
  }

  // The special run() method is invoked once upon object creation.
  // It specifies the object's active behavior.
  Unit run() {
    while (pingsLeft > 0) {
       pong ! hi("Hello");
       pingsLeft = pingsLeft - 1;
    }
  }
}
```

Interfaces. In ABS, classes are not types. Instead, all object references are typed by *interfaces*. An interface declaration consists of the interface name, a list of interfaces that the interface inherits from, a list of methods that have to be implemented by classes implementing the interface. All elements except the interface name are optional. A class has to implement at least the methods that are listed in its interface(s). These methods listed in the implemented interfaces can be called from the outside. All other methods the class implements are private and can only be invoked on `this`. Interfaces do not contain field declarations. Hence, there is no way of accessing fields of an object different from `this`.

Example:

```
interface Empty {
  Unit doNothing();
}

class IEmpty implements Empty {
  Unit doNothing() { skip; }
  Unit thisIsPrivate() { skip; }
}
```

Statements. ABS supports standard statements and expressions known from object-oriented languages, such as Java. These include: assignments, conditional statements and loops. The most basic statement is **skip**, which does nothing. While not particularly useful, it has its place in empty method bodies of abstract behavioral models which will be concretized later. Variables in ABS are defined in the usual way by giving a type, name and initial value. Variable names must begin with a lowercase letter, followed by a combination of letters, numbers and the underscore (_) character. Variable assignments consist of a left-hand side naming a variable and a right-hand side which can be any type-correct expression.

Example:

```
String x = "Hello";
x = x + " World!";
```

The conditional statement has the same syntax as in Java. The conditional expression must be of type `Bool` (i.e., evaluate to `True` or `False`), the consequent and optional alternate parts of the conditional statement are blocks which can introduce local variables.

Example:

```
if (contains(ingredients(juice), Banana)) {
  result = "I love bananas!";
} else {
  skip;
}
```

The while loop is standard as well, consisting of a Boolean expression and a block, to be evaluated until the expression evaluates to `False`.

Example:

```
while (True) {
  skip; // This loops forever.
}
```

Modules. An *ABS Model* is a set of *modules*, where each module is defined in an ABS file, which typically ends with `.abs`. A file can have multiple module definitions, but a single module must be completely defined in one file.

Modules define named scopes for declarations which can be interfaces, classes, or data types, and provide name spaces and a means for implementation hiding. All declarations defined in a module are by default hidden and cannot be used by other modules. In order to make declarations available to other modules, they have to be explicitly *exported*. In order to use declarations of other modules, they have to be explicitly *imported*. The following example shows how names can be exported and imported. Like Java packages, modules in ABS are *flat*. Even though module names are often made hierarchical by using periods, such a structure has no special meaning in ABS.

Example:

```
module Example.PingPong.Ping;
export IPing, Ping;
import Pong from Example.PingPong.Pong;
```

A module can have an optional *main block*, which defines how a system is started. The following module has a main block that creates an instance of `IPing` and calls its `start` method. See the following subsection for the **new cog** syntax.

Example:

```
module Example.PingPong;
import * from Example.PingPong.Ping;
import * from Example.PingPong.Pong;

{
  Pong pong = new cog IPong();
  Ping ping = new cog IPing(pong,5);
}
```

2.3 Concurrency Model

ABS is especially designed for modeling concurrent and distributed systems. The concurrency model of ABS is based on the concept of *Concurrent Object Groups* (COGs). A typical ABS system consists of multiple COGs at runtime. COGs can be regarded as autonomous, runtime components that are executed concurrently and share no state.

Concurrent Object Groups. A new COG is created by using the **new cog** expression. It takes as argument a class name, which is the class of the first object of the new COG. The result is a reference to the first object. The following example creates a new COG with an initial object of class `IPong`. The reference to this new `IPong` object is stored in the `pong` variable. The `IPong` object lives in a different COG than the COG which created it.

Example:

```
Pong pong = new cog IPong();
```

Asynchronous Method Calls. Objects communicate by exchanging messages via method calls. When using a synchronous method call, the caller must wait for the call to be returned. This leads to a strong temporal coupling between the caller and the callee. In a distributed setting, the caller must additionally also wait until the corresponding network message has been sent to the target node, which leads to problems for systems with high latency. ABS contains linguistic constructs for synchronous method calls. However, due to the above reasons, communication between COGs may solely be via *asynchronous method calls.* The difference to the synchronous case is that an asynchronous call immediately returns to the caller without waiting for the message to be received and handled by the callee. Asynchronous method calls are indicated by an exclamation mark (!) instead of a dot.

Example:

```
pong ! hi("Hello Pong");
```

In order to ensure that COGs only communicate via asynchronous methods calls, ABS provides a pluggable type extension to statically distinguish *far references* to objects in a different COG and *near references* in the same COG. The used type annotations are `[Near]`, `[Far]`, and `[Somewhere]`, where `[Somewhere]` means that the reference is either near or far. The above example can be then typed as follows.

Example:

```
[Far] Pong pong = new cog IPong();
```

As synchronous method calls are not allowed on far references the following code will lead to a runtime error in ABS. When using the additional type annotations, the type checker will catch that error at compile time already.

Example:

```
Pong pong = new cog IPong();
pong.hi("Hello"); // runtime error
```

Futures. Communication between objects usually follows a *request-response pattern*. If a request is sent via a synchronous method call, eventually the called object sends the return value of the call as return value to the caller. When using asynchronous method calls, the caller does not wait for the result of the call. Instead, the asynchronous method call returns a *future*. A future is a placeholder for the result of the method call. Initially, a future is *unresolved*. When the called method has terminated, the future will (automatically) be *resolved* with the result value of the call. The caller can, thus, obtain the result value of the method call at a later point by using the future.

A future in ABS is represented by the predefined data type **Fut**`<T>` where the type parameter `T` corresponds to the return type of the called method. The following example assigns the result of the above asynchronous method call to a future `answerFut`, where the method `hi` is assumed to have `String` as return type. To get the value of the future `answer`, the **get**-expression can be used.

Example:

```
Fut<String> answerFut = pong ! hi("Hello Pong");
String answer = answerFut.get;
```

The **get**-expression only returns the value of the future, when the future is resolved. If the future is unresolved, the control flow is *blocked*, until the future is resolved. Hence, synchronous communication can be simulated in ABS by performing an asynchronous method call and waiting for the resolved future using the **get**-expression.

Cooperative Multi-Tasking. The ABS approach to handle concurrency relies on strict data encapsulation and on cooperative multi-tasking on the level of COGs. Strict data encapsulation is achieved since all object fields are private and can only be accessed via method calls. Hence, the state of an object does not have to be protected against modifications from the outside.

COG-level cooperative multi-tasking means that all *tasks*[1] run within the scope of a COG. A method call creates a task in the scope of the target object. For asynchronous methods calls, the calling task can continue to run while the issued method call is processed and get its result at a later point in time. Race conditions between the tasks of the same COG are prevented by cooperative multi-tasking. For example, in conventional programming languages that are based on preemptively scheduled threads, the following code is prone to subtle errors:

Example:

```
Unit addToState(Int item) {
  itemCount = itemCount + 1;
  itemList = Cons(item, itemList);
}

Int removeFromState() {
  Int result = 0;
  if (itemCount > 0) {
    itemCount = itemCount - 1;
    result = head(itemList); itemList = removeHead(itemList);
  } else {
    // handle error
  }
  return item;
}
```

If a thread running `addToState` is interrupted after its first statement, a second thread running `removeFromState` might try to remove an element from an empty `itemList` which is a typical race condition. In languages like Java race conditions can only be prevented by explicitly synchronizing threads using locks or synchronized blocks.

ABS solves this problem by scheduling tasks only at specific *scheduling points* during program execution which are apparent in the source code. Hence, a COG state is implicitly protected, except at certain points that can be syntactically identified and analyzed. The **suspend** statement introduces a scheduling point, allowing the running task to be suspended and another task of the COG to be scheduled.

[1] Tasks correspond to threads, known from languages such as Java, but are scheduled cooperatively instead of preemptively.

Example:

```
// This loops forever, blocking the COG
while (True) { skip; }

// This loops forever, in parallel with other tasks
while (True) { suspend; }
```

With the **await** statement, one can create a conditional scheduling point, where the running task is suspended, until the specified condition becomes true.

Example:

```
Bool flag = False;

Bool waitUntilTrue () {
  await flag; // we rely on some other task to set the flag for us
  return flag; // will always return True
}
```

The **await** statement is also a way to synchronize with the future of an asynchronous method call without blocking the entire COG.

Example:

```
  Fut<String> answerFut = ping ! hi("Hello Ping");
  skip; // do some processing ...
  await answerFut?;
  String answer = answerFut.get; // guaranteed not to block
```

A method for reasoning about absence of race conditions in ABS is to inspect each **suspend** and **await** statement, and check if the task at this point leaves the COG in an orderly state (i.e., establishes the COG invariant). At all other points, the COG is implicitly protected against concurrent modifications.

3 Spatial Variability Modeling

Spatial variability captures different variants of a software products coexisting at the same point in time. This variability can often be phrased in terms of the *features* offered by the software. Finer-grained configuration parameters are represented using *attributes* of features. Software product line engineering [18] aims at developing similar product variants by reuse. The ABS incarnation of this approach is realised by four languages (μTVL, DML, CL, PSL) on top of core ABS. These languages express spatial variability at the level of product features and as changes to the behavior of a core product, along with providing the configuration of the product line artifacts, and the ultimate selection of a product via the specification of the relevant features and their attributes.

The feature description language μTVL is used to describe the variability of a product line in terms of features and their attributes. At this level of abstraction, a feature is just a name. Attributes represent micro-variability within features. DML is used to specify delta modules which, when selected, are used to modify a core ABS model. Delta modules implement spatial variability at the level of core ABS. Although DML delta modules are used to implement features, they are written independently of any feature. They are, in fact, reusable for different ABS product lines as they may be written independently of a specific application context. CL specifications link μTVL feature models with the DML delta modules that implement the corresponding behavioral modifications. A CL specification provides *application conditions* for each delta module, which are constraints over features and their attributes governing when the delta module is applicable. A CL specification also specifies constraints on the ordering of applicable delta modules, in order to avoid potential ambiguity in different delta application orders. A PSL script consists of two parts, namely, a specification of the features and their attributes selected for a product and an initialization code block, which typically is just a call to an appropriate *main* method, though it may contain additional configuration. The feature selection part of a PSL specification is checked against a μTVL feature model, and is also used to determine which delta modules to apply, namely, those whose application condition is true given the feature selection. To generate the product specified by the PSL script, all deltas with valid application condition are applied to the core ABS model in some order compliant with the order specified in the CL script, and then the initialisation block is added to the core program.

The following application, the core of a multilingual "Hello World" program, will be used to illustrate these languages and their interaction. Here is the core of the `MultiLingualHelloWorld` product line in core ABS:

Example:

```
interface Greeting {
  String say_hello();
}
class Greeter implements Greeting {
  String say_hello() {
    return "Hello world";
  }
}
class Application {
  Unit main() {
    Greeting bob;
    bob = new Greeter();
    String s = "";
    s = bob.say_hello();
  }
}
```

Interface `Greeting`, class `Greeter` and class `Application` form the core modules of the ABS implementation.

3.1 Feature Modeling

μTVL is a feature modelling language, pronounced either *micro textual variability language* or simply *mu tee vee ell*. It is an extended subset of TVL [5,6], which was developed to serve as a reference language for specifying feature models. μTVL is textual, as opposed to diagrammatic, and aims to be scalable, concise, modular, and comprehensive. A feature model is represented textually as a forest of nested features, each with a collection of boolean or integer attributes. Additional cross-tree dependencies can be expressed in the feature model. μTVL allows a feature model with multiple roots (hence, multiple trees) to express orthogonal variability [18], which is useful for representing application or platform models in an orthogonal fashion.

The grammar of μTVL is given in Fig. 2. FID is the set of valid feature names, and AID of valid attribute names. Attributes and values in μTVL range either over integers or over booleans. Extensions to include other data types is unproblematic, as long as any relevant constraints can be encoded by integer constraints.

```
           Model ::= (root FeatureDecl)* FeatureExtension*

     FeatureDecl ::= FID [{ [Group] AttributeDecl* Constraint* }]
FeatureExtension ::= extension FID { AttributeDecl* Constraint*}

           Group ::= group Cardinality { [opt] FeatureDecl, ([opt] FeatureDecl)* }
     Cardinality ::= allof | oneof | [n1 .. *] | [n1 .. n2]
   AttributeDecl ::= Int AID ; | Int AID in [ Limit .. Limit ] ; | Bool AID ;
           Limit ::= n | *

      Constraint ::= Expr ; | ifin: Expr ; | ifout: Expr ;
                   | require: FID ; | exclude: FID ;
            Expr ::= true | false | n | FID | AID | FID.AID
                   | UnOp Expr | Expr BinOp Expr | ( Expr )
            UnOp ::= ! | -
           BinOp ::= || | && | -> | <-> | == | != | > | < | >= | <= | + | - | * | / | %
```

Fig. 2. Syntax of μTVL (n ranges over integers)

The *Model* clause specifies a number of "orthogonal" root feature models and some extensions. A root feature model, *FeatureDecl*, contains the name of a feature (FID), followed by a specification of any sub-features, the feature's attributes and any relevant constraints. Extensions, *FeatureExtension*, specify additional attributes and constraints, typically cross-tree dependencies. The *Group* clause specifies the sub-features of a feature. This consists of a specification of the cardinality of the group, plus a number of possibly optional sub-features. The *Cardinality* clause describes the number of elements of a group that may appear in a result. Keyword `allof` means that all elements of the group must appear. Keyword `oneof` means that one element must appear. Range descriptions `[`n_1 `..` $*$`]` and `[`n_1 `..` n_2`]` specify the range of values on the number of elements of the group. These can be bounded below and above or unbounded above ($*$). Zero or one instances of each feature can be present in the ultimate model—this means that cardinality specifies not that features can be multiply instantiated, rather it specifies the number of selections that can be made for a choice: by analogy, $\{Apple, Banana\}$ is a valid choice of 2 elements from the set $\{Apple, Orange, Banana\}$, whereas $\{Orange, Orange\}$ is not.

The *AttributeDecl* clause specifies both integer (bounded or unbounded) and boolean attributes of features. The *Limit* clause is used to specify the bounds, where n is some integer and $*$ indicates that an attribute is unbounded below and/or above.

The *Constraint* clause specifies constraints on the presence of features and on attribute values. An `ifin` constraint is only applicable if the current feature is selected. Similarly, an `ifout` constraint is only applicable if the current feature is not selected. An `include` clause specifies that the current feature requires some other feature, whereas `exclude` expresses the mutual incompatibility between the current feature and some other feature. The *Expr* clause ultimately expresses a boolean constraint over the presence of features and attribute values. Features are referred to by identity (FID). Attributes are referred to either using an unqualified name (AID), for in scope attributes, or using a qualified name (FID.AID) for attributes of other features. Unary, *UnOp*, and binary operators, *BinOP*, are standard.

Example 1. The following is a feature model for the `MultiLingualHelloWorld` product line, which describes software that outputs *"hello world"* in multiple languages some number of times.

```
root MultiLingualHelloWorld {
  group allof {
    Language {
      group oneof { English, Dutch, French, German }
    },
    opt Repeat {
      Int times in [0..1000];
      times > 0;
    }
  }
}

extension English {
  ifin: Repeat ->
       (Repeat.times >= 2 && Repeat.times <= 5);
}
```

This feature model introduces two core features, `Language` and `Repeat`. The `Language` feature corresponds to one of four possible features: `English`, `Dutch`, `French`, or `German`. The `Repeat` feature has no sub-features, and it has an attribute `times` with range from 0 to 1000, with an added condition that it must be strictly greater than 0. The extension for the `English` feature states that when the `English` and `Repeat` features are both present, the attribute `times` must be between 2 and 5, inclusive.

This feature model can be depicted as a feature diagram using standard notations [7], as shown in Fig. 3.

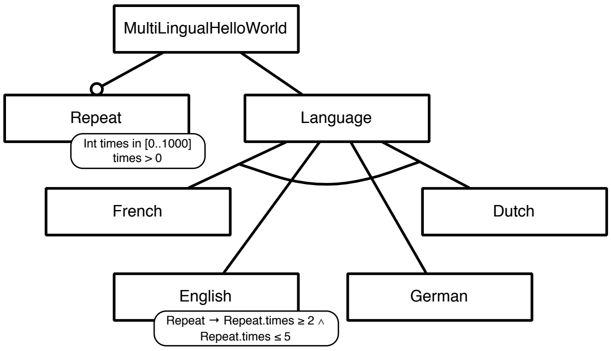


Fig. 3. Feature Diagram for the *MultiLingualHelloWorld* example

3.2 Delta Modeling

Variability at the level of abstract behavioral specifications (or source code) is achieved using delta modeling. The concept of delta modeling was introduced by Schaefer et al. [19,20,22,21] as a novel modeling and programming language approach for software-based product lines, and can be seen as an direct alternative to feature-oriented programming [2]. Both approaches aim at automatically generating software products for a given valid collection of features, providing flexible and modular techniques to build different products that share functionality or code. In delta-oriented programming [20], *application conditions*, conditions over the set of features and their attributes, are associated with modules of program modifications (add, remove, or otherwise modify code), called *delta modules*. The implementation of a software product line in delta-oriented programming is divided into a *core module* and a set of delta modules. The core module consists of the classes that implement a complete product of the corresponding product line. Delta modules describe how to change the core module to obtain new products. The choice of which delta modules to apply is based on the selection of desired features for the final product. For representing spatial variability in the ABS language, we adapt these ideas to ABS models.

Fig. 4 specifies the syntax of delta modules over core ABS models. The grammar uses nonterminals from the core ABS language, indicated in purple (gray). Their names should be sufficiently suggestive.

```
              DeltaDecl ::= delta TypeId [DeltaParamDecls]
                                      {ClassOrInterfaceModifier* }
ClassOrInterfaceModifier ::= adds ClassDecl
                           | modifies class TypeId ImplementsModifier*
                                      { Modifier* }
                           | removes class TypeId ;
                           | adds InterfaceDecl
                           | modifies interface TypeId ImplementsModifier*
                                      { Modifier* }
                           | removes interface TypeId ;

      ImplementsModifier ::= adds TypeId
                           | removes TypeId

                Modifier ::= adds FieldDecl
                           | removes FieldDecl
                           | adds MethDecl
                           | modifies MethDecl
                           | removes MethSig

         DeltaParamDecls ::= (DeltaParamDecl (, DeltaParamDecl)* )
          DeltaParamDecl ::= Identifier HasCondition*
                           | Type Identifier

            HasCondition ::= hasField FieldDecl
                           | hasMethod MethSig
                           | hasInterface TypeId
```

Fig. 4. Syntax of Delta Modules

The *DeltaDecl* clause specifies the syntax of delta modules, which consists of an unique identifier, a list of parameters, and a body containing a sequence of class and interface modifiers. The *ClassOrInterfaceModifier* clause describes the syntax of modifications at the level of classes and interfaces. Such a modification can add a class or interface declaration, modify an existing class or interface, or remove a class or interface. The *ImplementsModifier* clause describes how to modify the interfaces a class implements or extends, either by adding new or removing existing interfaces.

The *Modifier* clause specifies the modifications that can occur within a class or interface body. These include (where relevant) adding and removing fields and method signatures (from interfaces), and modifying methods, which amounts to replacing a method with a new implementation, but enabling the original method to be called using the **original**`()` keyword. The semantics of calling **original**`()` is essentially the same as `Super()` from feature-oriented programming [2], and `proceed` from context-oriented programming [13], and similar to ordinary `super` calls in standard object-oriented languages, as well as the `around` advice (without quantification) from aspect-oriented programming [16].

Delta modules in the ABS language can be parameterised both by attribute values and by class names, to enable the application of a single delta in more than one context. This is in contrast to delta modules presented in the literature [20,22,21], which are unparameterised. The *HasCondition* describes constraints on class arguments to which a delta may be applied. These constraints state, for instance, the methods and fields that a class or interface is expected to have.

Below are some delta modules for the `MultiLingualHelloWorld` product line defined above which represent the `Repeat`, `German` and `Dutch` features. A delta module for the `French` feature can be implemented in a similar fashion.

Example:

```
delta Rpt (Int times) {
  modifies class Greeter {
    modifies String say_hello() {
      String result = "";
        Int i = 0;
        while (i < times) {
          result = result + original();
          i = i + 1;
        }
        return result;
     }
  }
}
delta De {
  modifies class Greeter {
    modifies String say_hello() {
      return "Hallo Welt";
    }
  }
}
delta Nl {
  modifies class Greeter {
    modifies String say_hello() {
      return "Hallo wereld";
    }
  }
}
```

The delta module `De`, for example, implements a single class modifier for `Greeter`, which in turn has a single method modifier. This method modifier replaces the method `say_hello()` of class `Greeter` to return the German text "Hallo Welt". The delta module `Rpt` has a single parameter for the number of times that the hello string should be repeated. This delta module replaces the method `say_hello()` in class `Greeter` with new ABS code. However, in this case, the code being replaced is also included via the special method call **original**`()`.

3.3 Product Line Configuration

The product line configuration language (CL) links μTVL feature models with DML delta modules to provide a complete specification of the spatial variability in an ABS product line [22,21]. A product line configuration script consists of the set of features assumed to exist and a set of *delta clauses*. Each delta clause names a delta module and specifies the conditions required for its application, called *application conditions*. A partial ordering on delta modules specified by after clauses constrains the order in which delta modules can be applied to the core module. The syntax of the product line configuration language is given in Fig. 5.

```
         Configuration ::= productline Name { Features ; DeltaClauses }

              Features ::= features FID ( , FID )*

          DeltaClauses ::= DeltaClause ( , DeltaClause)*
           DeltaClause ::= delta DeltaSpec [AfterCondition] [ApplicationCondition] ;

             DeltaSpec ::= Name [( DeltaArguments )]
        DeltaArguments ::= DeltaArgument ( , DeltaArgument)*
         DeltaArgument ::= FID | FID.AID | PureExpr

        AfterCondition ::= after Name ( , Name )*
  ApplicationCondition ::= when PureExpr
```

Fig. 5. CL grammar

The *Configuration* clause specifies the name of the product line, the set of features it provides using a *Features* clause, and the set of delta modules used to implement those features by *DeltaClause* clauses. In a *DeltaClause*, the *AfterCondition* clause specifies the delta modules that the current delta module must be applied after. The *DeltaSpec* clause names a specific delta module and specifies any parameters that need to be passed. These parameters are either attributes from the feature model or constant values specified in core ABS as a *PureExpr*, that is, an expression without side effects. The *ApplicationCondition* clause specifies a predicate describing under which feature configurations the given delta module is applied. This condition is phrased in terms of the presence and absence of features and feature combinations, as well as using attributes of features and integer and boolean constants.

The `MultiLingualHelloWorld` product line is configured as follows:

Example:

```
productline MultiLingualHelloWorld {
  features English, German, French, Dutch, Repeat;

  delta Rpt(Repeat.times) after De, Fr, Nl when Repeat;
  delta De when German;
  delta Fr when French;
  delta Nl when Dutch;
}
```

The example CL configuration script specifies, for example, that the `De` delta module is applicable when the `German` feature is selected. Note that there is no delta module corresponding to the feature `English`, as the core provides support for the English feature as a default. In addition, `Rpt` is configured such that it has to be applied **after** all the language-specific delta modules. The delta module `Rpt`'s argument is filled with the `times` attribute of feature `Repeat`, which ultimately will be specified in a PSL script (Section 3.4).

3.4 Product Selection and Generation

To generate a product from an ABS product line, a product selection is specified using the *product selection language* (PSL). A product selection states which features are to be included in the product and specifies concrete values for their attributes. In addition, some core ABS code is provided to initialise the selected product. A product selection is checked against a μTVL feature model for validity, possibly after adding any implied features. An implied feature is a parent feature in the μTVL feature model. Then, the product selection is used by the configuration file to guide the application of the delta modules during the generation of the final software product. Fig. 6 specifies the grammar of PSL.

```
            Selection ::= product Name ( FeatureSpecs ) { InitBlock }

         FeatureSpecs ::= FeatureSpec (, FeatureSpec)*
          FeatureSpec ::= FID [AttributeAssignments]

 AttributeAssignments ::= { AttributeAssignment (, AttributeAssignment)* }
  AttributeAssignment ::= AID = Value

            InitBlock ::= { Core ABS code }
```

Fig. 6. PSL grammar

The *Selection* clause specifies a product by giving it a name, by stating the features (*FeatureSpec*) to be included in the product and the concrete values for their attributes (*AttributeAssignment*) and by specifying an initialisation block (*InitBlock*). This can be any core ABS code, but typically, it will be a simple call to some already present *main* method. Initialisation blocks are specified in the product selection language to enable product lines with multiple entry points to the code base.

Here are some candidate product selections for the `MultiLingualHelloWorld` product line:

Example:

```
// basic product with no deltas
product P1 (English) {
  Application.main();
}

// apply delta Fr
product P2 (French) {
  Application.main();
}

// apply deltas De and Repeat
product P3 (German, Repeat{times=10}) {
  Application.main();
}

// apply delta Repeat to core with feature English, but the application should
// be refused because "times > 5"
product P4 (English, Repeat{times=6}) {
  Application.main();
}
```

The example specifies four products: `P1`, `P2`, `P3`, and `P4`. In the case of the product `P1`, the parameter `English` means the product consists of this feature and of the features implied by the feature model. In this case, the implied features are `Language` and the root `MultiLingualHelloWorld`, according to the feature model in Example 1. In `P3` and `P4`, the parameters also include attribute values, in these cases assigning a value to the attribute `times` of the `Repeat` feature. The block of ABS code associated to each product provides its initialisation code. Every product in our example executes the main method of the class `Application`, which is included in the core module.

3.5 Product Generation

Given a core ABS program P, a set of delta modules Δ, a product line configuration C, a feature model FM, and a product selection p, the final software product, which will be a core ABS program, is derived as follows:

- first, check that the product selection p satisfies the constraints imposed by the feature model FM;
- second, select the delta modules from Δ with a valid application condition with respect to p;
- third, apply the delta modules to the core program P in some order respecting the partial order described in C; and
- finally, add the initialisation block to the resulting ABS code.

The selection of product `P3` in our running example results in the following Core ABS model. The parameter `times` from feature `Rpt` has been replaced by the value 10, as specified in the product selection. Class `Application` and interface `Greeting` are as above in the core module. The **original**`()` call to the previous version of the `say_hello` method is replaced with a call to a renamed version of the unchanged method.

Example:

```
class Greeter implements Greeting {
  String say_hello_original() {
    return "Hallo Welt";
  }
  String say_hello() {
    String result = "";
    Int i = 0;
    while (i < 10) {
      result = result + say_hello_original();
      i = i + 1;
    }
    return result;
  }
}
{ // initialisation block
  Application.main();
}
```

4 Temporal Variability Modeling

The basic idea of temporal variability modeling in ABS is to capture how products in an ABS product line safely evolve over time. Evolution takes place during system execution in order to accomodate necessary changes after the deployment of the products; e.g., bug fixes, feature extensions or modifications, or changes in user requirements. The main design target of ABS models are concurrent and distributed systems in which objects communicate asynchronously. Consequently, it is not straightforward to halt a deployed product during execution in order to let it evolve. Rather, temporal evolution must happen asynchronously at runtime. In ABS, we follow an approach developed for class-based inheritance for distributed concurrent objects [14], but we adapt it to delta modeling.

4.1 Dynamic Delta Modules

In order to facilitate the modeling of temporal variability for high-level ABS models, it is crucial that evolution is expressed at the abstraction level of the modeling language. Therefore, temporal variability is captured by a *series of asynchronous changes* to the executing product, where each change addresses one of the structuring concepts of the modeling language and where the series of changes together bring about the desired overall modification of product behavior. In ABS, the main structuring concepts for system variability are a designated core module and delta modules, which contain modifier operations for classes, interfaces, field and method declarations (see Fig. 4).

```
           DynDeltaDecl ::= dyndelta TypeId [DeltaParamDecls]
                                 { (DeltaModifier | ClassOrInterfaceModifier)* }

          DeltaModifier ::= adds delta DeltaDecl
                          | modifies delta DeltaDecl

              DeltaDecl ::= TypeId [DeltaParamDecls]
                                      {ClassOrInterfaceModifier* }

ClassOrInterfaceModifier ::= adds ClassDecl
                          | adds InterfaceDecl
                          | modifies class TypeId ImplementsModifier*
                                      { Modifier* }
                          | simplifies class TypeId
                                      { Simplifier* }

     ImplementsModifier ::= adds TypeId

               Modifier ::= adds FieldDecl
                          | adds MethDecl
                          | modifies MethDecl

             Simplifier ::= removes FieldDecl
                          | removes MethDecl
```

Fig. 7. Syntax for dynamic delta declarations in ABS (the clause for *DeltaParamDecls* is defined in Fig. 4)

Temporal variability in ABS is expressed in terms of *dynamic delta modules*. Dynamic delta modules can add classes or interfaces to the core module or modify classes or interfaces that are already contained in the core module. Additionally, dynamic delta modules can change delta modules, meaning they can add classes or interfaces which are in turn added by the delta modules or alter the contained modification operations. The grammar for dynamic delta modules is given in Fig. 7. Dynamic delta modules allow us to add new class and interface declarations to the core module or delta modules, and to change existing class declarations in the following ways:

– add new fields and method definitions
– modify existing method definitions
– simplify the class by removing redundant field and method declarations

Thus, the dynamic delta modules are slightly more restrictive than standard delta modules used for spatial variability. In particular, classes and interfaces cannot be removed, and the modifiers are distinguished from the simplifiers (in Fig. 4, the simplifiers are in the same syntactic category as the modifiers).

In order to illustrate the usage of dynamic delta modules, the `MultiLingual HelloWorld` product line is modified to make a more personal greeting. The dynamic delta module `ExtendedGreeting` changes the `Greeter` class that is defined in the core module by adding a method `i_am_bob()` which returns a personalized message and modifies the `say_hello()` method to output this message. The delta modules `De` and `Nl` are modified to include a translated version of the `i_am_bob()` method:

Example:

```
dyndelta ExtendedGreeting {
  modifies class Greeter { // Adds a new method to the class Greeter
    adds String i_am_bob () { return ", I am Bob!"; }
    modifies String say_hello() {
         return = original() + this.i_am_bob();
     }
   }

modifies delta De {
 modifies class Greeter {
  modifies String i_am_bob() {return ", ich bin Bob!";}
  }
 }

modifies delta Nl {
 modifies class Greeter {
  modifies String i_am_bob() {return ", ick ben Bob!";}
  }
 }
}
```

4.2 Restrictions on Dynamic Delta Modules

The rationale for the syntactic restrictions introduced in the dynamic delta modules is to guarantee that the temporal evolution of the product's code does not give rise to runtime errors. This guarantee is provided by a static analysis which identifies runtime applicability conditions for the different elements of the dynamic delta module. In order to apply a change to a class, other changes may be required to have taken place already. In the asynchronous setting of ABS, this cannot be statically controlled. For example, if new code in a change to a class D invokes a method on an instance of another class C, that method must be available. If the method is introduced as a change to C, the static analysis of the change to D will generate an applicability condition to require that this change to C has already occurred at runtime. The distinction between modifiers and simplifiers corresponds to the distinction between runtime applicability conditions at the level of classes and at the level of instances of those classes [14]. The removal of classes, as well as the removal and modification of interfaces, are disallowed because they would require a inspectionvof all references of the involved interfaces (including references in messages), which is a very heavy operation in the asynchronous distributed setting targeted by ABS.

5 Case Study

In this section, we present a case study of spatial and temporal variability modeling with the ABS based on the Fredhopper Access Server (FAS) [11]. In particular, we focus on its Replication System component. First, we describe how we model the core components of the existing Java implementation of the FAS replication system using Core ABS. Second, we capture some of the system's spatial variabilities using the languages presented in Section 3. Third, we illustrate how to represent temporal variability, including changing of available features. Finally, we describe how the HATS tools suite assists our modeling activities.

5.1 Fredhopper Access Server

The Fredhopper Access Server (FAS) is a service-oriented, server-based software system, which provides search and merchandising IT services to e-Commerce companies, such as large catalogue trading, travel booking, or classified advertising, etc. Each FAS installation is deployed to a customer according to the FAS deployment architecture. Fig. 8 shows an example setup.

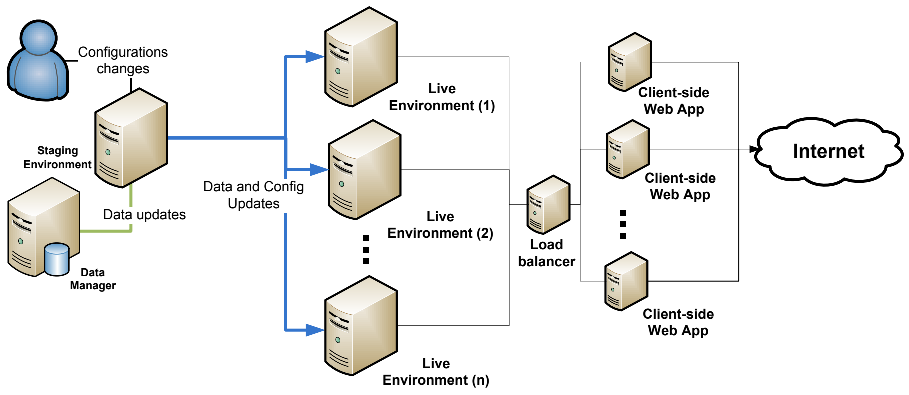


Fig. 8. Example of FAS deployment

A FAS deployment consists of a set of live environments and a staging environment. A live environment processes queries from client web applications via web services. The staging environment is responsible for receiving data updates in XML format, indexing the XML, and distributing the resulting indices across all live environments according to the replication protocol. A more detailed description of the replication system can be found in [8].

For the purpose of this case study, we focus on the synchronisation client component in the live environment and the synchronisation server component in the staging environment. The synchronisation server and a set of synchronisation clients together constitute the replication system. The synchronization server distributes configuration and data updates to the synchronization clients running on the live environments. It is responsible for determining the schedule of replication as well as the content of each replication item for its staging environment. A replication item represents a single unit of replicable data. It is either a file directory, a set of files whose name matches a regular expression or a database journal. A replication snapshot is a set of replication items. The synchronisation client connects to the synchronisation server component in the staging environment and responds to incoming updates resulting from changes to data and configuration.

The synchronisation server and clients do not communicate directly. The synchronisation server creates connection threads that serve as the interface to the server-side of the replication protocol. In the existing implementation, connection threads are Java thread objects. The synchronisation client, on the other hand, schedules client jobs to handle communications to the client-side of the replication protocol. In the existing implementation, a client job is a Java object that is scheduled using a third party scheduling library. The synchronisation server and clients communicate via connection threads and client jobs asynchronously via sockets.

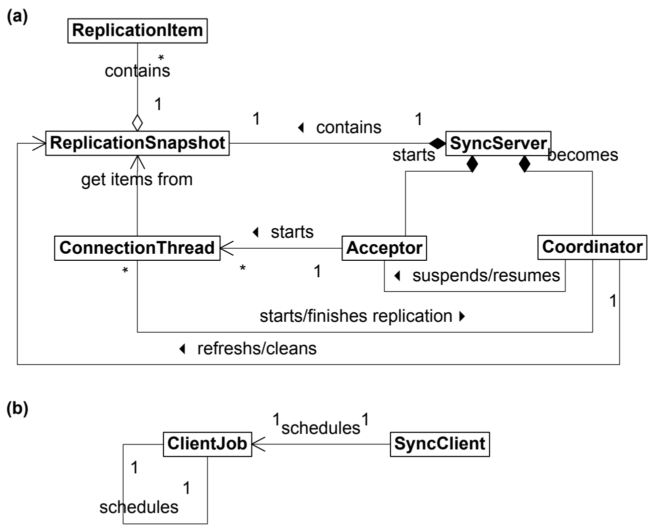


Fig. 9. Class diagram of (a) synchronisation server and (b) synchronisation client

Fig. 9(a) and (b) show the UML class diagram of the synchronisation server and client respectively. The synchronisation server consists of the following components: an acceptor, one or more connection threads, a coordinator, a SyncServer and a replication snapshot. The synchronisation client, on the other hand consists of a SyncClient and one or more client jobs.

5.2 Modeling the Replication System with Core ABS

We now provide a description of the individual components of the replication system in our ABS model. Listing 1 shows the ABS interfaces for the components of the synchronisation server. For brevity, we have omitted ABS class definitions. More details on the ABS model described in this section can be found on the HATS project website `http://www.hats-project-eu`.

```
command(Command command); } interface ConnectionThread extends
Command { } interface Node { DataBase getDataBase(); } interface
ServerNode extends Node { Set<Schedule> getSchedule(); }

interface Acceptor {
  [Far] ConnectionThread getConnection(ClientJob job);
  Bool isAcceptingConnection();
  Unit suspendConnection();
  Unit resumingConnection(); }

interface Coordinator {
  Unit process();
  Unit startUpdate(ConnectionThread worker);
  Unit finishUpdate(ConnectionThread worker); }

interface SyncServer extends ServerNode {
  Acceptor getAcceptor();
  [Far] Coordinator getCoordinator();
  [Near] ReplicationSnapshot getReplicationSnapshot(); }

interface ReplicationSnapshot {
  Unit refreshSnapshot(Bool refreshSnapshot);
  Unit clearSnapshot();
  Int getIndexingId();
  Set<ReplicationItem> getItems();
  Bool hasUpdated(); }

interface ReplicationItem {
  FileEntry getContents();
  ReplicationItemType getType();
  FileId getAbsoluteDir();
  Unit refresh();
  Unit cleanup(); }
```

Listing 1. ABS interfaces of the synchronisation server components

The *Acceptor* component is responsible for accepting connections from the synchronisation clients and is specified by interface `Acceptor` shown in Listing 1. The interface provides a method for a client job to obtain a reference to a connection thread, as well as methods to enable and disable the synchronisation server to accept a new client job connection. The connection thread and the client job are specified by interfaces `ConnectionThread` and `ClientJob` in Listing 1 and Listing 2, respectively.

```
interface SyncClient extends Node {
  [Far] Acceptor getAcceptor();
  ClientDataBase getClientDataBase();
  Unit becomesState(State state);
  Unit setAcceptor(Acceptor acceptor); }

interface ClientJob extends Command {
  Bool registerReplicationItems(CheckPoint checkpoint);
  Maybe<FileSize> processFile(FileId id);
  Unit processContent(File file);
  Unit receiveSchedule(); }
```

Listing 2. ABS interfaces of the synchronisation client components

Each *connection thread* is instantiated by the `Acceptor`. After the `Acceptor` accepts a connection from a client job, it instantiates a `ConnectionThread` to carry out the replication protocol. The connection thread is specified by interface `ConnectionThread` shown in Listing 1. `ConnectionThread` exposes a single method `command()`, which is asynchronously invoked by `ClientJob` objects to determine the current state of a replication.

The *Coordinator* is responsible for coordinating when the `Acceptor` may accept connections from synchronisation clients. This component also provides methods for preparing replication items before a replication session and clearing them afterwards. The coordinator is specified by interface `Coordinator` shown in Listing 1.

The *SyncServer* starts the `Acceptor` and the `Coordinator`. It also keeps a reference to the relevant replication snapshot. The synchronization server is specified by the interface `SyncServer` shown in Listing 1.

Listing 2 shows the ABS interfaces of the components that are part of the synchronization client. The *SyncClient* communicates with the `SyncServer` via job scheduling. At initialisation time, the `SyncClient` schedules a client job to acquire a replication schedule from the server. Using this schedule, this client job creates a new client job for performing the actual replication. Each client job thereafter is responsible to request replication schedules and set up the subsequent jobs for further replication.

Each *client job* receives replication items from a connection thread and updates the synchronisation client's files (configuration and data). The client job is specified by interface `ClientJob` shown on Listing 2. Client jobs may be scheduled either sequentially or concurrently.

Listing 3 shows an example main block of the replication system. In this main block, first an exemplary set of changes to the data in the synchronisation server is defined. In our ABS model, the file system is structured as a tree, where non-leaf nodes are directories. The changes to a file directory are specified by the map `items`, whose keys are `CheckPoint` values identifying particular sets of changes. Each key points to a set of `File` values representing updates to those files. A file is identified by its fully qualified name, specifying the path through the directory tree to the file content. As simplification, the file content is denoted by an integer value representing its size. For example, in the listing the key `1` points to updates on files located at `dir1/file` and `dir2/file2`.

```
{
Map<CheckPoint,Map<FileId,FileContent>> items =
 map[Pair(1,map[file("dir1/file1",1),file("dir2/file2",2)]),..];

Set<Schedule> schedules =
 set[FileItem("dir2","dir2/dir21"),..];

Set<ClientId> cids = set[0,..];
Set<[Far] SyncClient> syncclients = EmptySet;
Set<ClientId> iterator = cids;
while (hasNext(iterator)) {
  Pair<Set<ClientId>,ClientId> nt = next(iterator);
  SyncClient syncclient = new cog SyncClientImpl(snd(nt));
  syncclients = insertElement(syncclients,syncclient);
  iterator = fst(nt); }

SyncServer syncserver =
  new cog SyncServerImpl(items,schedules);

Fut<Acceptor> acc = syncserver!getAcceptor(); await acc?;
[Far] Acceptor acceptor = acc.get;

Set<SyncClient> clientIterator = syncclients;
while (hasNext(clientIterator)) {
  Pair<Set<SyncClient>,SyncClient> nt = next(clientIterator);
  SyncClient syncclient = snd(nt);
  syncclient!setAcceptor(acceptor);
  clientIterator = fst(nt); }
}
```

Listing 3. An example main block

The main block defines a set of schedules for replication and instantiates a corresponding set of synchronisation clients, each as a separate COG. Additionally, a synchronisation server is instantiated in another COG, and a reference to its acceptor class is obtained. Afterwards, the main block passes this reference to the clients, which triggers the replication protocol. By instantiating all clients and the server as separate COGs, all connection threads and client jobs belong to separate COGs and, thus, can only communicate via asynchronous method calls.

5.3 Spatial Variability of the Replication System

The replication system can exist in several variants. Listing 4 shows the corresponding feature model. The replication system has a feature `JobProcessing`, which requires an alternative choice between the two features `Seq` and `Concur`, capturing the choice between sequential and concurrent client job processing, respectively. Additionally, the replication system has a feature *ReplicationItem* which allows choosing between three replication item types represented by the features `Dir`, `File` and `Journal`. The `Dir` feature is mandatory, that is, all versions of the replication system support replicating complete file directories. Moreover, the `Journal` feature requires the feature `Seq` which means that variants of the replication system that support database journal replication may only schedule client jobs sequentially.

```
root ReplicationSystem {
  group allof {
    JobProcessing {
      group oneof { opt Seq, opt Concur }
    },
    ReplicationItem {
      group [1..*] {
        Dir, opt File, opt Journal { require: Seq; }
      }
    }
  }
}
```

Listing 4. Feature model of the replication system in μTVL

The core model of the replication system supports sequential client job processing. This functionality is implemented by the active class `ClientJobImpl`. A partial ABS class definition of `ClientJobImpl` is shown in Listing 5. Each instance of `ClientJob` initialises the Boolean field `newJob` to `False` and invokes its `run` method. This method in turn invokes `scheduleNewJob()` asynchronously. The method `scheduleNewJob()` waits for field `newJob` to become `True` before creating a new instance of `ClientJob`. Setting `newJob` to `True` at the end of the run method ensures that each client job is scheduled sequentially. The method becomeState() is invoked synchronously at specific points inside the run method to ensure that, while scheduling client jobs sequentially, the synchronisation client follows a predefined *client state machine* [8].

The lower half of Listing 5 defines the delta module Concurrent which specifies a single class modifier for class ClientJobImpl that has two method modifiers. The first modifier removes the await statement from scheduleNewJob() in such a way that a new instance of ClientJob is created as soon as the current ClientJob instance releases the lock of this object group. This potentially allows scheduling client jobs concurrently. The second modifier updates method becomeState so that the synchronisation client is not required to follow the client state machine which only applies to sequential scheduling.

```
class ClientJobImpl([Far] InternalClient client, JobType job)
implements ClientJob {
  Bool newJob = False;
  Unit scheduleNewJob() {
    await newJob;
    new ClientJobImpl(this.client, Replication);
  }
  Unit run() { .. this!scheduleNewJob(); .. newJob = True; .. }
  Unit becomeState(State state) { .. }
  ..
}

delta Concurrent {
  modifies class ClientJobImpl {
    modifies Unit scheduleNewJob() {
      new ClientJobImpl(this.client, Replication);
    }
    modifies Unit becomeState(State state) { .. }
  }
}
```

Listing 5. Core module and delta module for job processing

Listing 6 shows a partial definition of the classes DirectoryItem and ReplicationSnapshotImpl. The class DirectoryItem defines a replication item for a complete file directory and the class ReplicationSnapshotImpl implements ReplicationSnapshot. The method replicationItem defined in the class ReplicationSnapshotImpl takes a replication schedule, creates a corresponding ReplicationItem object and adds it to the set of replication items. By default, this method only handles replication schedules for complete file directories.

```
class DirectoryItem(FileId qualified, ServerDataBase db)
implements ReplicationItem { .. }

class ReplicationSnapshotImpl(
  ServerDataBase db, Set<Schedule> schedules)
  implements ReplicationSnapshot {

  Set<ReplicationItem> items = EmptySet;

  Unit replicationItem(Schedule schedule) {
    if (isSearchItem(schedule)) {
     FileId qualified = left(item(schedule));
     ReplicationItem item = new DirectoryItem(qualified,this.db);
     this.items = Insert(item,this.items);
    }
  }
  ..
}
```

Listing 6. Partial core implementation of replication item

In Listing 7, two delta modules are depicted that implement the necessary functionality for other types of replication items. The delta module `File` is applied for handling file set replication and has two class modifiers. The first modifier adds class `FilePattern`, an implementation of interface `ReplicationItem` handling replicating file sets that matches a regular expression. The second modifier updates the method `replicationItem` to handle replication schedules with file sets. The delta module `Journal` contains the necessary modifications for handling database journal replication. It has two class modifiers to add a new implementation of interface `ReplicationItem` and to update the method `replicationItem` to handle replication schedules with data base journals.

```
delta File {
  adds class FilePattern(FileId qualified, String pattern,
                         ServerDataBase db)
    implements ReplicationItem { .. }

  modifies class ReplicationSnapshotImpl {
    modifies Unit replicationItem(Schedule schedule) {
      original();
      if (isFileItem(schedule)) {
        Pair<FileId,String> it = right(item(schedule));
        ReplicationItem item =
          new FilePattern(fst(it),snd(it),this.db);
        items = Insert(item,items);
      }
    }
  }
}

delta Journal {
  adds class JournalItem(FileId qualified, ServerDataBase db)
    implements ReplicationItem { .. }

  modifies class ReplicationSnapshotImpl {
    modifies Unit replicationItem(Schedule schedule) {
      original();
      if (isJournalItem(schedule)) {
       FileId qualified = left(item(schedule));
       ReplicationItem item = new JournalItem(qualified,this.db);
       this.items = Insert(item,this.items);
      }
    }
  }
}
```

Listing 7. Delta modules for replication items

Listing 8 shows the product line configuration of the replication system in CL, where the features `Dir` and `Seq` are the features provided by the core module. The application condition for delta `Concurrent` states that this delta is applied if and only if feature `Concur` is selected and feature `Journal` is not selected. This application condition respects the constraint specified in the feature model shown in Listing 4.

```
productline ReplicationSystem {
  features Dir, File, Journal, Seq, Concur;
  delta File when File;
  delta Journal when Journal;
  delta Concurrent when Concur && (~ Journal);
}

product DS (Dir, Seq) { .. } // default product (core)
product DFC (Dir, File, Concur) { .. } // file pattern, concurrent
product DC (Dir, Concur) { .. } // directory, concurrent
product DFJS (Dir, File, Journal, Seq) { .. } // directory, concurrent
```

Listing 8. Product line configuration and product selections for the replication system

Listing 8 also shows some example product selections for the replication system product line specified in PSL. For brevity, details of the main blocks have been omitted. For example, product `DS` defines a variant of the replication system that supports the core set of features. Product `DFJS` is a variant that supports all types of replication items, and product `DFC` supports both directory and file set replication, as well as concurrent client job scheduling.

5.4 Temporal Variability of the Replication System

Existing products have to evolve to meet the market demand for new features. Thus, a product line may have to be changed simultaneously in several dimensions, which makes the management of the evolution a difficult task. A complete re-modeling of an evolved product line has a high cost. Therefore, it is beneficial to re-use model artifacts from previous versions of the product line in a compositional and incremental manner.

As an evolution from the current versions, Fredhopper aims to develop a loosely coupled, pluggable architecture for FAS. This architecture will allow components to be added and removed from a FAS deployment dynamically at runtime. One component that will benefit from this pluggable architecture is the Search Engine Optimizer (SEO) component. Search engine optimization improves the visibility of a client's website in search engines via search results. The SEO component includes an indexing facility for these search results which must also be replicated from the staging environment across all live environments.

The replication system currently supports replicating complete directories, file sets, and database journals. In order to add support for replication of SEO search result indices, we use dynamic delta modules to change the ABS model of the replication system product line. The dynamic delta module `IndexItemDelta` modifies the replication system such that an SEO component can be added to a FAS deployment at runtime. Listing 9 shows a partial implementation of this dynamic delta module. Read from top to bottom, the dynamic delta module `IndexItemDelta` in Listing 9 provides the following modifications:

1. Addition of an interface `Indexer` to model a generic indexer.
2. Addition of a class `SEOIndexer` implementing the `Indexer` interface for the SEO indexing facility. For brevity, implementation details are omitted.
3. Addition of the interface `IndexItem` that extends the interface `Replication-Item` to associate an `Indexer` to a replication item.
4. Modification of the class `ReplicationSnapshotImpl` so that method `refreshSnapshot()` creates a new instance of the class `Indexer`, before refreshing individual items.
5. Modification of the delta module `File` in the following ways:

   (a) Modification of the class `ReplicationSnapshotImpl` (which is already modified by the delta module `File`) in such a way that the method `createReplicationItems()` associates the indexer with file set replication items.

(b) Modification of the class `FilePattern` (that is added by the delta module `File`) such that it implements the interface`IndexItem` and such that the method `refresh()`, which refreshes the files contained in the replication item, acquires a set of indices (`Set<Index>`) stored at the file locations pointed to by the replication item. In the ABS model, we represent indices by the algebraic data type `Index`.

```
dyndelta IndexItemDelta {

  adds interface Indexer { // adding new indexer interface
    Set<Index> getIndex(FileId id, String pattern);
  }

  // a default implementation of the indexer
  adds class SEOIndexer(ServerDataBase db)
  implements Indexer { .. }

  // extends replication item to handle indices
  adds interface IndexItem extends ReplicationItem {
     Unit setIndexer(Indexer indexer);
  }

  modifies class ReplicationSnapshotImpl { // adds indexing support
    adds Indexer indexer = null;
    modifies Unit refreshSnapshot(Bool refreshSnapshot) {
      this.indexer = new SEOIndexer(this.db);
      original();
    }
  }

  modifies delta File { // adds indexing support
    modifies class ReplicationSnapshotImpl {
      modifies Unit createReplicationItems() {
        original();
        if (isFileItem(schedule)) {
          Pair<FileId,String> it = right(item(schedule));
          IndexItem item =
            new FilePattern(fst(it),snd(it),this.db);
          item.setIndexer(indexer);
          items = Insert(item,items);
        }
      }
    }
    modifies class FilePattern adds IndexItem {
      adds Set<Index> indices = EmptySet;
      adds Indexer indexer = null;
      adds Unit setIndexer(Indexer i) { this.indexer = i; }
      modifies Unit refresh() {
        original();
        this.indices = indexer.getIndex(qualified,pattern);
      }
    }
  }
}
```

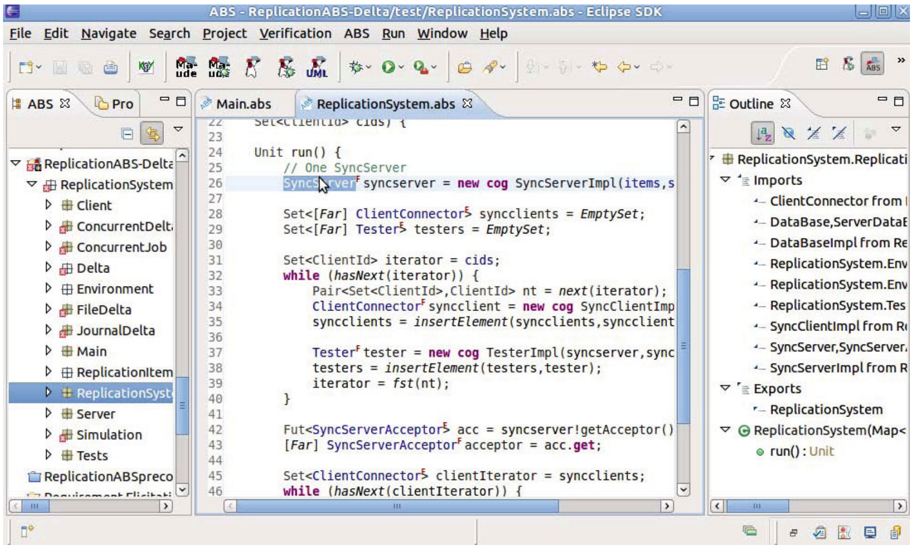
Listing 9. Dynamic delta `IndexItemDelta` for supporting the SEO component

The dynamic delta module `IndexItemDelta` only provides support for search result indices used by SEO components. However, it can be composed with another delta module that provides a different implementation of the interface `Indexer`, thereby reusing dynamic delta module `IndexItemDelta` to support the indexing facility of other pluggable components.

6 ABS Tool Suite

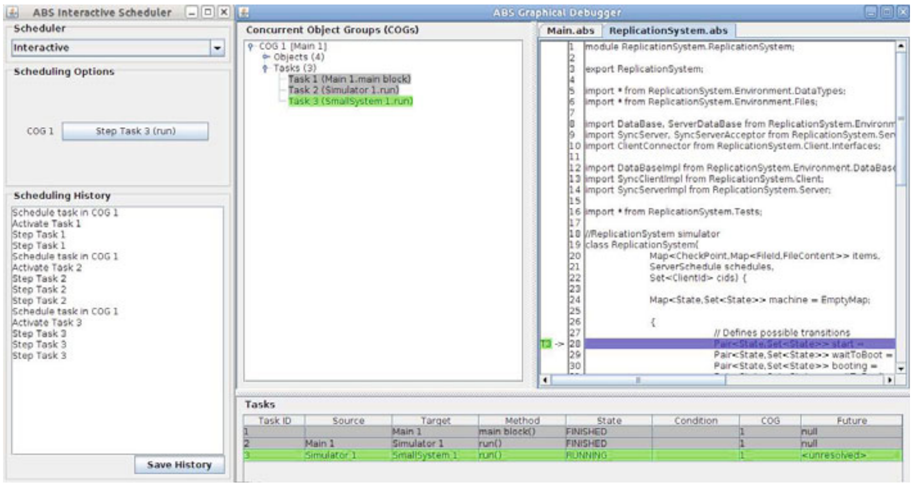
The ABS tool suite supports the modeling of highly configurable, distributed and concurrent systems in ABS. Specifically, the ABS compiler front-end takes a full ABS model as input, which includes the core ABS model in the Core ABS language, a feature model descriptions in μTVL, delta modules in DML, a product line configuration in CL, product selections in PSL. It checks the model for syntax and semantic errors and translates it into an internal representation. Several different back-ends can then be used to translate the internal representation into different languages, like Maude and Java, which allows ABS models to be executed and analyzed on these platforms.

Moreover, the HATS framework comes with a plug-in for Eclipse, a graphical debugger and a sequence diagram visualizer. The plug-in provides an Eclipse perspective for navigating, editing, parsing and type checking ABS models. It also provides integrations with different back-ends, allowing the generation of both Java and Maude code from ABS models and the execution and debugging of the generated code directly in the Eclipse perspective. Fig. 10 shows a screen shot of the Eclipse perspective, taken when conducting the replication system case study. The left hand pane in the figure shows the ABS module view. In this view, ABS modules, imports/exports, classes, interfaces, functions and data types of an open ABS Eclipse project are shown, independent of the actual ABS (.abs) files where they are defined in. The middle pane depicts the ABS editor view for editing ABS (.abs) files. The editor provides syntax highlighting, jump-to-declaration, content assistance, and code completion. The right hand pane is



```
    Set<ClientId> cids) {

    Unit run() {
        // One SyncServer
        SyncServer syncserver = new cog SyncServerImpl(items,s

        Set<[Far] ClientConnector> syncclients = EmptySet;
        Set<[Far] Tester> testers = EmptySet;

        Set<ClientId> iterator = cids;
        while (hasNext(iterator)) {
            Pair<Set<ClientId>,ClientId> nt = next(iterator);
            ClientConnector syncclient = new cog SyncClientImp
            syncclients = insertElement(syncclients,syncclient

            Tester tester = new cog TesterImpl(syncserver,sync
            testers = insertElement(testers,tester);
            iterator = fst(nt);
        }

        Fut<SyncServerAcceptor> acc = syncserver!getAcceptor()
        [Far] SyncServerAcceptor acceptor = acc.get;

        Set<ClientConnector> clientIterator = syncclients;
        while (hasNext(clientIterator)) {
```

Fig. 10. ABS Eclipse Perspective



```
Schedule task in COG 1
Activate Task 1
Step Task 1
Step Task 1
Schedule task in COG 1
Activate Task 2
Step Task 2
Step Task 2
Step Task 2
Schedule task in COG 1
Activate Task 3
Step Task 3
Step Task 3
Step Task 3
```

```
module ReplicationSystem.ReplicationSystem;

export ReplicationSystem;

import * from ReplicationSystem.Environment.DataTypes;
import * from ReplicationSystem.Environment.Files;

import DataBase, ServerDataBase from ReplicationSystem.Environm
import SyncServer, SyncServerAcceptor from ReplicationSystem.Ser
import ClientConnector from ReplicationSystem.Client.Interfaces;

import DataBaseImpl from ReplicationSystem.Environment.DataBase
import SyncClientImpl from ReplicationSystem.Client;
import SyncServerImpl from ReplicationSystem.Server;

import * from ReplicationSystem.Tests;

//ReplicationSystem simulator
class ReplicationSystem(
            Map<CheckPoint,Map<FileId,FileContent>> items,
            ServerSchedule schedules,
            Set<ClientId> cids) {

            Map<State,Set<State>> machine = EmptyMap;

            {
                        // Defines possible transitions
                        Pair<State,Set<State>> start =
                        Pair<State,Set<State>> waitToBoot =
                        Pair<State,Set<State>> booting =
```

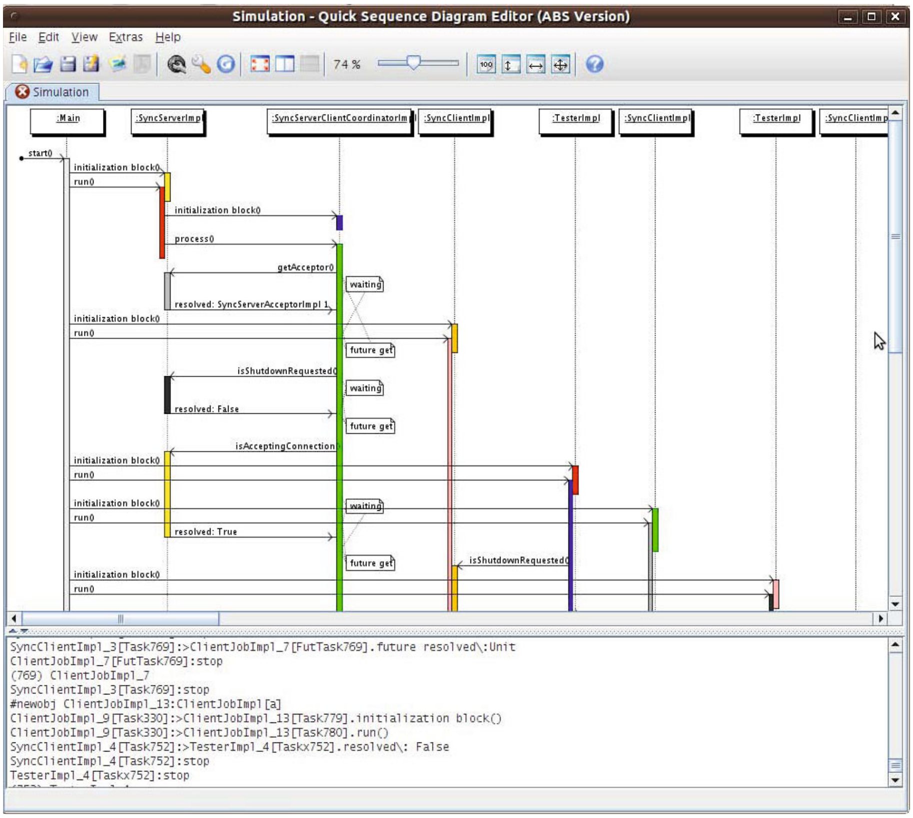
Task ID	Source	Target	Method	State	Condition	COG	Future
1		Main 1	main block()	FINISHED		1	null
2	Main 1	Simulator 1	run()	FINISHED		1	null
3	Simulator 1	SmallSystem 1	run()	RUNNING		1	<unresolved>

Fig. 11. ABS Graphical Debugger

the ABS outline view, which shows the structure of the active ABS file. It is updated as the user edits the file.

Fig. 11 shows a screen shot of the ABS graphical debugger, taken when conducting the replication system case study. The graphical debugger allows stepping through ABS models, either interactively or via a seeded random sequence of steps. The left hand pane in the figure shows the current scheduler and the sequence of steps that have already been executed. The right pane shows the current state of the execution. The window includes the state of individual concurrent object groups, a list of concurrent tasks, and an ABS model view, relating active tasks to the currently executing statement in the ABS model.

Fig. 12 shows a screen shot of the ABS sequence diagram visualizer, taken when conducting the replication system case study. The visualizer is used in conjunction with the graphical debugger. When stepping through ABS models via the debugger, the visualizer shows the asynchronous messages sent between objects in various concurrent object groups.



```
SyncClientImpl_3[Task769]:>ClientJobImpl_7[FutTask769].future resolved\:Unit
ClientJobImpl_7[FutTask769]:stop
(769) ClientJobImpl_7
SyncClientImpl_3[Task769]:stop
#newobj ClientJobImpl_13:ClientJobImpl[a]
ClientJobImpl_9[Task330]:>ClientJobImpl_13[Task779].initialization block()
ClientJobImpl_9[Task330]:>ClientJobImpl_13[Task780].run()
SyncClientImpl_4[Task752]:>TesterImpl_4[Taskx752].resolved\: False
SyncClientImpl_4[Task752]:stop
TesterImpl_4[Taskx752]:stop
```

Fig. 12. ABS Sequence Diagram Visualizer

In the course of the HATS project, the development of the tool suite and the case studies go hand-in-hand. On the one hand, the tool suite encourages the practical application of the ABS. This is evidently shown by modeling a large piece of production code, such as the replication system, with the ABS. On the other hand, the continuous application of the ABS to real-life case studies steers the development of the ABS tool suite ever closer to an industrial-strength framework.

7 Conclusion

The ABS modeling language of the HATS project, together with the HATS ABS tool suite constitutes a new model-based approach to the development of concurrent systems that exhibit a high degree of variability.

The Core ABS language (Section 2) is a state-of-the-art, strongly typed, abstract, concurrent, object-based modeling language that is fully executable. Tightly coupled groups of objects with shared memory are encapsulated into *concurrent object groups* (COGs) that realize data-race free computation via cooperative multi-tasking. Objects from different COGs may only use asynchronous communication and message passing. Future types make it possible to continue computation, while waiting for the result of an asynchronous method call. Functional expressions over parametric algebraic data types are used to model data in an implementation-independent way.

A novelty of ABS is the possibility to define software product lines with the help of feature models and delta modeling. Delta modeling is a flexible code reuse technique that has been shown to be particularly suitable for the description of product lines (see [9,20] for a thorough discussion). In contrast to architectural languages, ABS provides a formal link between features and their implementation. A dynamic version of delta modules allows describing the evolution of ABS models at runtime (Section 4).

As illustrated in Section 6, the ABS language is integrated into a tool suite that includes editing, parsing, type checking, compilation (into Java and Maude), debugging, and visualization. The ABS language has a formal, mathematical semantics which is the basis of advanced tools, partially under development, that address test case generation, model mining, functional verification, various static analyses, resource analysis, etc.[2] The aim of HATS is to position ABS and its tool suite as a *single source* technology for architectural modeling, functional modeling, verification, and implementation of highly configurable, concurrent software systems [23].

The case study in Section 5 demonstrates that this is not just a vision, but a realistic goal. We illustrated how to model a core component of a complex, industrial distributed system using the ABS language. Specifically, we have shown:

– How to use the core ABS language to model the concurrent aspects of the replication system.
– How to refine the core model using the ABS language to express important variabilities of the replication system to capture all possible component variants.
– How to capture evolution (variabilities over time) with the ABS language in order to reduce modeling cost.

The HATS tool suite, documentation, as well as several case studies, including the one discussed here, are available from `http://www.hats-project.eu`.

[2] A description of these is beyond the scope of this tutorial.

References

1. Atkinson, C., Bayer, J., Bunse, C., Kamsties, E., Laitenberger, O., Laqua, R., Muthig, D., Paech, B., Wüst, J., Zettel, J.: Component-based product line engineering with UML. Addison-Wesley Longman Publishing Co., Inc., Boston (2002)
2. Batory, D., Sarvela, J., Rauschmayer, A.: Scaling Step-Wise Refinement. IEEE Trans. Software Eng. 30(6) (2004)
3. Beck, K.: Extreme Programming. Addison-Wesley, Reading (1999)
4. de Boer, F.S., Clarke, D., Johnsen, E.B.: A complete guide to the future. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 316–330. Springer, Heidelberg (2007)
5. Boucher, Q., Classen, A., Faber, P., Heymans, P.: Introducing TVL, a text-based feature modelling language. In: Proceedings of the Fourth International Workshop on Variability Modelling of Software-intensive Systems (VaMoS 2010), Linz, Austria, January 27-29, pp. 159–162. University of Duisburg-Essen (2010), `http://www.vamos-workshop.net/2010`
6. Classen, A., Boucher, Q., Heymans, P.: A text-based approach to feature modelling: Syntax and semantics of TVL. Science of Computer Programming (November 2010), `http://linkinghub.elsevier.com/retrieve/pii/S0167642310001899`
7. Czarnecki, K., Eisenecker, U.: Generative programming. Addison-Wesley, Reading (2000)
8. Evaluation of Core Framework (August 2010), deliverable 5.2 of project FP7-231620 (HATS), `http://www.hats-project.eu`
9. Final Report on Feature Selection and Integration (March 2011), deliverable 2.2b of project FP7-231620 (HATS), `http://www.hats-project.eu`
10. Full ABS Modeling Framework (March 2011), deliverable 1.2 of project FP7-231620 (HATS), `http://www.hats-project.eu`
11. Fredhopper Access Server, `http://www.fredhopper.com`
12. Gomaa, H.: Designing Software Product Lines with UML. Addison Wesley, Reading (2004)
13. Hirschfeld, R., Costanza, P., Nierstrasz, O.: Context-oriented Programming. Journal of Object Technology (March/April 2008)
14. Johnsen, E.B., Kyas, M., Yu, I.C.: Dynamic classes: Modular asynchronous evolution of distributed concurrent objects. In: Cavalcanti, A., Dams, D. (eds.) FM 2009. LNCS, vol. 5850, pp. 596–611. Springer, Heidelberg (2009)
15. Johnsen, E.B., Owe, O.: An asynchronous communication model for distributed concurrent objects. Software and System Modeling 6(1), 35–58 (2007)
16. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J., Irwin, J.: Aspect-oriented programming. In: Liu, Y., Auletta, V. (eds.) ECOOP 1997. LNCS, vol. 1241. Springer, Heidelberg (1997)
17. Medvidovic, N., Taylor, R.: A Classification and Comparison Framework for Software Architecture Description Languages. IEEE Transactions on Software Engineering (2000)
18. Pohl, K., Böckle, G., Van Der Linden, F.: Software Product Line Engineering: Foundations, Principles, and Techniques. Springer, Heidelberg (2005)
19. Schaefer, I.: Variability Modelling for Model-Driven Development of Software Product Lines. In: Proc. of 4th Intl. Workshop on Variability Modelling of Software-intensive Systems (VaMoS 2010) (2010)
20. Schaefer, I., Bettini, L., Bono, V., Damiani, F., Tanzarella, N.: Delta-oriented programming of software product lines. In: Bosch, J., Lee, J. (eds.) SPLC 2010. LNCS, vol. 6287, pp. 77–91. Springer, Heidelberg (2010)

21. Schaefer, I., Bettini, L., Damiani, F.: Compositional Type-Checking for Delta-oriented Programming. In: Intl. Conference on Aspect-oriented Software Development (AOSD 2011) (2011, to appear)
22. Schaefer, I., Damiani, F.: Pure Delta-oriented Programming. In: FOSD 2010 (2010)
23. Schaefer, I., Hähnle, R.: Formal methods in software product line engineering. IEEE Computer 44(2), 82–85 (2011)
24. Schäfer, J., Poetzsch-Heffter, A.: JCoBox: Generalizing active objects to concurrent components. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 275–299. Springer, Heidelberg (2010)
25. Ziadi, T., Hélouët, L., Jézéquel, J.M.: Towards a UML Profile for Software Product Lines. In: van der Linden, F.J. (ed.) PFE 2003. LNCS, vol. 3014, pp. 129–139. Springer, Heidelberg (2004)

Kernel-Based Machines for Abstract and Easy Modeling of Automatic Learning

Alessandro Moschitti

Computer Science and Engineering Department,
University of Trento,
Via Sommarive 18, Povo (TN), Italy
moschitti@disi.unitn.it
http://disi.unitn.it/moschitti

Abstract. The modeling of system semantics (in several ICT domains) by means of pattern analysis or relational learning is a product of latest results in statistical learning theory. For example, the modeling of natural language semantics expressed by text, images, speech in information search (e.g. Google, Yahoo,..) or DNA sequence labeling in Bioinformatics represent distinguished cases of successful use of statistical machine learning. The reason of this success is due to the ability to overcome the concrete limitations of logic/rule-based approaches to semantic modeling: although, from a knowledge engineer perspective, rules are natural methods to encode system semantics, noise, ambiguity and errors affecting dynamic systems, prevent such approached from being effective, e.g. they are not flexible enough.

In contrast, statistical relational learning, applied to representations of system states, i.e. training examples, can produce semantic models of system behavior based on a large number attributes. As the values of the latter are automatically learned, they reflect the flexibility of statistical settings and the overall model is robust to unexpected system condition changes. Unfortunately, while attribute weight and their relations with other attributes can be automatically learned from examples, their design for representing the target object (e.g. a system state) has to be manually carry out. This requires expertise, intuition and deep knowledge about the expected system behavior. A typical difficult task is for example the conversion of structures into attribute-value representations.

Kernel Methods are powerful techniques designed within the statistical learning theory. They can be used in learning algorithms in place of attributes, thus simplifying object representation. More specifically, kernel functions can define structural and semantic similarities between objects (e.g. states) at abstract level, replacing the similarity defined in terms of attribute overlap.

In this chapter, we provide the basic notions of machine learning along with latest theoretical results obtained in recent years. First, we show traditional and simple machine learning algorithms based on attribute-value representations and probability notions such as the Naive Bayes and the Decision Tree classifiers. Second, we introduce the PAC learning theory and the Perceptron algorithm to provide the readers with essential concepts of modern machine learning. Finally, we use the above background to illustrate a simplified theory of Support Vector Machines, which, along with the kernel methods, are the ultimate product of the statistical learning theory.

1 What Is Machine Learning?

In high school, in the mathematic or statistic classes, we have been taught techniques that, given a set of points, e.g. $\boldsymbol{x}_i$ and the values associated with them, i.e. y_i, attempt to derive the functions that best interpolates their relationships $\phi(\boldsymbol{x}_i, y)$. For example, linear or polynomial regression as shown in Figure 1. These techniques, e.g. least square fit, are the first examples of machine learning algorithms. When the output values, y_i, of the target function are finite and discrete, the regression problem is called classification, which is very interesting for the application on real scenarios, e.g. categorization of text documents in different topics.

Before introducing more advanced machine learning techniques, it is helpful to show an example of their usefulness in ICT. Let us suppose that a programmer is asked to write the following program: given some employee characteristics and a pre-defined employee level hierarchy, automatically assign to each new employee the adequate entry level. Moreover, suppose that (i) the rules to determine such entry level depends on many variables, e.g. achieved diplomas, previous working experiences, age and so on; and (ii) there is no formal document that explains how to produce such rule set. This is not an unrealistic situation as the target company may use such level information to only propose tasks to employees; thus the level may be heuristically assigned by the human resource department by using an informal algorithm.

The unlucky programmer would be soon in troubles as it is rather difficult to extract algorithmic information from people not used to think in terms of procedures and instructions. What might be the solution?

We note that, there is a lot of data about the link between variables (i.e. the employees) and the output of the target function (i.e. the entry level). The company keeps the data of employees along with their entry levels, thus the programmer may examine the data and try to hand-craft the rules from it. However, if the number of employees and the number of their characteristics are large, this would result in a very time consuming and boring task.

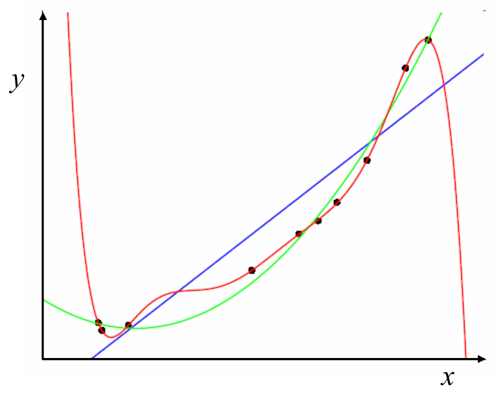


Fig. 1. Polynomial interpolation of as set of points $\langle x_i, y_i \rangle$

Machines have traditionally been built to perform such kind of job, thus, the solution should rely on writing an algorithm which automatically derives from examples the employee classification rules. This kind of algorithms are a special class of *machine learning methods* called example-driven or inductive learning models. They are standard in the sense that they can be applied to all problems in which there are some data examples and we need a classification function as output.

Given such tools, the lucky programmer can only re-write the examples from the employee database in an input format suitable for the target machine learning algorithm and run it to derive the classification function. The latter function unlikely will provide a correct entry level in all cases but if the commissioning company (as in this case) accepts an certain error rate in this procedure, the application of an automatic approach will be a feasible alternative to the hand-coding. Indeed, another output of the learning process is usually the expected error rate. This value can be estimated by measuring the number of classification mistakes that the classification function commits on a set of employee data (test set) not used for training.

We have introduced what learning models may offer to the solution of real problems. In the next section, we illustrate two simple ML approaches based on Decision Trees and naive probabilistic models. These will clarify the importance of kernel methods for more quickly and easily define the appropriate learning system.

1.1 Decision Trees

The introduction has shown that ML models derive a classification function from a set of training examples (e.g. the employee data) already categorized in the target class (e.g. the entry level). The input for the ML program is the set of examples encoded in a meaningful way with respect to the classification task, i.e. the level assignment. The variables describing the individual examples are usually called features and they capture important aspects of the classification objects, e.g. the employees. For instance, the study title is a relevant feature for the entry level whereas the preferred employee food is not relevant thus it should not be included in the example description.

The idea of decision tree classifier (DT) algorithm is inspired by a simple principle: the feature that correctly separates the highest number of training examples should be used before the others. To simplify such idea, suppose that we have only two levels (0 and 1) and also the features are binary (e.g. the employee has or not a master degrees). Intuitively, the decision tree algorithm finds the feature which splits the training set S in two subsets S_0 and S_1 such that the proportion of employees of level 0 is *higher* in S_0 than in S whereas the proportion of employees of level 1 is higher in S_1 than in S. This means that guessing the employee level in the two new sets is *easier* than in S. As we cannot hope to correctly separate all data with only one feature the algorithm will iteratively find other features that *best* separates S_0 and S_1.

Figure 2 illustrates the decision tree which a DT algorithm may generate. First, the *PhD* attribute is tested. In case the employee owns it the level is surely 1. Second, features such as *Previous Experiences* and *Intelligent Quotient* are tested. Finally, the tests on the leaves should output the final classification in case it had not been output on the internal nodes.

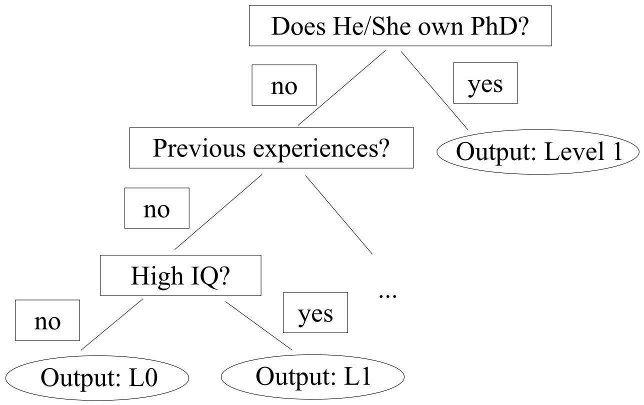


Fig. 2. Decision tree generated for the classification task of two employee levels

In order to find the most discriminative feature, DTs use the *entropy* quantity. In the general case, we have a set of classes $\{C_1, .., C_m\}$ distributed in the training set S with the probabilities $P(C_i)$, then the entropy H of P is the following:

$$H(P) = \sum_{i=1}^{m} -P(C_i)log_2(P(C_i)) \tag{1}$$

Suppose to select a feature f which assumes $\{a_1, .., a_n\}$ values in S. If we split the training examples according to f, we obtain n different subsets, i.e. $\{S_1, .., S_n\}$, whose average entropy is:

$$\bar{H}(P^{S_1}, .., P^{S_n}) = \sum_{i=1}^{m} \frac{H(P^{S_i})}{|S_i|} \tag{2}$$

where, P^{S_i} is the probability distribution of C_i on the S_i set and $H(P^{S_i})$ is the related entropy.

The DT algorithm evaluates Eq. 2 for each feature and selects the one which is associated with the highest value. Such approach uses the probability theory to select the most informative features and generate the tree of decisions. In the next section, we show another machine learning approach in which the probability theory is more explicitly applied for the design of the decision function.

1.2 Naive Bayes

We have pointed out that machine learning approaches are useful when the information about the target classification function (e.g. the commissioned program) is not explicitly available and is not completely accurate. Such aspects determine a degree of uncertainty, which results in an error rate.

Given the random nature of the expected outcome, the probability theory is well suited for the design of a classification function that aims to achieve the highest probability in producing correct results. Indeed, we can model the output of our target function as the probability to categorize an instance in a target class, given some parameters estimated from the training data.

More formally, let us indicate with E the classification example and let $\{C_1, .., C_m\}$ be the set of categories in which we want to classify such example. We are interested to evaluate the probability that E belongs to C_i, i.e. $P(C_i|E)$. In other words, we know the classifying example and we need to know its category. Our example E can be represented as a set of features $\{f_1, .., f_n\}$ but we do not know how to relate $P(C_i|f_1, .., f_n)$ to the training examples. Thus, we can use the Bayes' rule to derive a more useful probability form:

$$P(C_i|f_1, .., f_n) = \frac{P(f_1, .., f_n|C_i) \times P(C_i)}{P(f_1, .., f_n)}, \tag{3}$$

where

$$\sum_{i=1}^{m} P(C_i|f_1, .., f_n) = \sum_{i=1}^{m} \frac{P(f_1, .., f_n|C_i) \times P(C_i)}{P(f_1, .., f_n)} = 1$$

for definition of probability.

We will choose for the example E the category C_i associated with the maximum $P(C_i|E)$. To evaluate such probabilities, we need to select a category i and count the number of examples that contain the whole set of features, $\{f_1, .., f_n\}$. Considering that in real scenarios, a training set may contain no more than 10,000 examples, we will unlikely be able to derive reliable statistics as n binary features determine 2^n different examples[1]. Thus, to make the Bayesian approach practical, we naively assume that features are independent. Given such assumption, Eq. 3 can be rewritten as:

$$P(C_i|f_1, .., f_n) = \prod_{k=1}^{n} \frac{P(f_k|C_i) \times P(C_i)}{P(f_1, .., f_n)} \tag{4}$$

As $P(f_1, .., f_n)$ is the same for each i, we do not need it to determine the category associated with the maximal probability. The $P(C_i)$ can be computed by simply counting the number of training examples labeled as C_i, i.e. $|C_i|$ and divide it by the total number of examples in all categories:

$$P(C_i) = \frac{|C_i|}{\sum_{j=1}^{m} |C_j|}$$

To estimate $P(f_k|C_i)$, we derive n_{ik}, i.e. the number of examples categorized as C_i that contain the feature f_k and we divide it by the C_i cardinality, i.e.

$$P(f_k|C_i) = \frac{n_{ik}}{|C_i|}$$

[1] If we assume uniform distribution, to have a chance that a target example of only 20 features is included in the training set, the latter has to have a size larger than 1 billion of examples.

Table 1. Probability distribution of $sneeze$, $cough$ and $fever$ features inside the Allergy, Cold and Healthy categories

Prob.	Allergy	Cold	Healthy
$P(C_i)$	0.05	0.05	0.9
$P(sneeze\|C_i)$	0.9	0.9	0.1
$P(cough\|C_i)$	0.7	0.8	0.1
$P(fever\|C_i)$	0.4	0.7	0.01

As an example of naive Bayesian classification suppose that we divide the healthy conditions of target patients in thee different categories: Allergy, Cold and Healthy. The features that we use to categorize such states are $f_1 = sneeze$, $f_2 = cough$ and $f_3 = fever$. Suppose also that we can derive the probability distribution of Table 1 from a medical database, in which f_1, f_2 and f_3 are annotated for each patient.

If we extract from our target patient the following feature representation $E = \{sneeze, cough, \sim fever\}$, where $\sim$ stands for not $fever$, we can evaluate his/her probabilities to be in each category i:

- P(Allergy $|E)$ = (0.05)(0.9)(0.7)(0.6)/P(E)=0.019/$P(E)$
- P(Cold $|E) = (0.05)(0.9)(0.8)(0.3)/P(E) = 0.01/P(E)$
- P(Healthy $|E) = (0.9)(0.1)(0.1)(0.99)/P(E) = 0.0089/P(E)$

According to the above table, the patient should be affected by allergy.

It is worth to note that such probabilities depend on the product of the probabilities of each feature. It may occur, especially when the training corpus is too small, that some of them never appear in the training examples of some categories. As a consequence, the estimation of the probability of a feature f in the category C_i, $P(f|C_i)$, will be 0. This causes the product of Eq. 4 of a category i to be 0, although the contributions of the other features in the product may be high. In general, assigning a probability equal 0 to a feature is a rough approximation as the real probability is just too small to be observed in the training data, i.e. we do not have enough data to find an occurrence of f.

To solve the above problem, smoothing techniques are applied. The idea is to give to the features which do not appear in the training data a small probability, α. To keep constant the overall feature probability mass, to the other features will be subtracted a small portion, β, of their probability such that the overall summation is still 1.

The simplest of such techniques is called Laplace smoothing. The new feature probability is the following:

$$P(f_k|C_i) = \frac{n_{ik} + a \times p_k}{|C_i| + a}$$

where p_k is a probability distribution and a is the size of a hypothetical set of examples, where we assume to have observed p_k. When we do not know any information about the not observed features, it is logical to assume a uniform distribution, i.e. $p_k = 1/a$ therefore $a = n$ and

$$P(f_k|C_i) = \frac{n_{ik} + 1}{|C_i| + n}.$$

The smoothing techniques improve the Naive Bayes model by providing a better estimation of the probability of the features not observed in the data. However, the independence assumption seems a serious limitation to the accuracy reachable by such approach. The next section illustrates more recent machine learning techniques, which do not need to make such assumptions. These are called Support Vector Machines and also offer the possibility to model object with abstract feature representations.

Exercise 1. Classify using a Naive Bayes learning algorithm and the probabilities in Table 1 all 8 possible examples, e.g. $\{sneeze, cough, \sim fever\}$, $\{sneeze, \sim cough, fever\}$,...

Exercise 2. Modify the probabilities in Table 1 to classify e.g. $\{sneeze, cough, \sim fever\}$ in class Cold with a Naive Bayes classifier.

Exercise 3. Define a new learning application using the Naive Bayes algorithm.

2 Probably Approximately Correct (PAC) Learning

So far, we have seen two different ML approaches, i.e. DT and Naive Bayes. They can be both applied to training examples to learn classification functions and estimate their accuracy on a test set of new examples. Intuitively, we may think that as the number of training examples increases the accuracy increases as well. Unfortunately, this is not generally true. For example, if we want to learn the difference between Allergy and Cold categories using only the $sneeze$ and $cough$ features, we will never reach high accuracy, no matter how many training examples we have available. This happens because such features do not deterministically separate the two classes.

Given such problems, we need some analytical results that helps us to determine (1) if our learning function is *adequate* for the target learning problem and (2) the probability of error according to the number of available training examples. The class of functions for which we have such analytical data is called the probably approximately correct (PAC) class.

The statistical learning theory provides mathematical tool to determine if a class of functions is PAC learnable. At the base of such result there is a new statistical quantity designed by two scientists, Vapnik and Chervonenkis, called VC-dimension. This gives a measure of the learning complexity and can be used to estimate the classification error.

In the next sections, we formally define the PAC function class, provide a practical example to derive the error probability of PAC functions and introduce the VC-dimension, which automatizes the estimation of such error.

2.1 Formal PAC Definition

The aim of ML is to learn some functions from a set (Tr) of training examples. These latter can be seen as data points that are associated with some discrete values $\mathcal{C} = \{C_1, .., C_n\}$, in case of classification problems or real number $\mathbb{R}$, in case of regression problem. We focus only on classification problems, i.e. on finding a function $f : X \rightarrow \mathcal{C}$ using $Tr \in X$. In general, the training examples are randomly drawn thus we need to deal with a probability distribution D on X.

The function f can be learned by using an algorithm, which can generate only a small subset of all possible functions. Such algorithm derives a function $h \in H$ from the examples, where H is the class of all possible hypotheses (functions) derivable with it. This suggests that h will hardly be equal to f, consequently, it is very useful to define a measure of its error.

A reasonable measure is the percentage of points for which f and h differ, i.e. the probability that given an example x, $P[f(x) \neq h(x)]$. Note that D is particularly important. As a trivial example, if the probability $D(x_0)$ of an element $x_0 \in X$ is 1 and $f(x_0) = h(x_0)$, the error rate will be 0, independently of the number of $x \in Tr$ for which $f(x) \neq h(x)$.

The above case is very rare and does not occur in practical situations. On the contrary, there is a large class of functions whose error decreases as the number of training examples increases. These constitute the PAC learnable functions. Their formal definition is the following:

- Let the function $f : X \to \mathcal{C}$ belongs to the class F, i.e. $f \in F$, where X is the domain and $\mathcal{C}$ is the codomain of f.
- Suppose that the training and the test documents $x \in X$ are generated with a probability D.
- Let $h \in H$ be the function that we learned from the examples provided that we can learn only functions in H, i.e. in the hypothesis space.
- The error of h, $error(h)$, is defined as $P[f(x) \neq h(x)]$, i.e. the percentage of miss-classified examples.
- Let m be the size of the training set, then F is a class of PAC learnable functions if there is a learning algorithm such that:
  - $\forall f \in F, \forall D \in X$ and $\forall \epsilon > 0, \delta < 1$
  - $\exists m$ such that $P[error(h) > \epsilon] < \delta$, i.e. the probability that the h's error is greater than ϵ is lower than δ.

In other words, a class of functions F is PAC learnable if we can find a learning algorithm which, given an enough number of training examples, produces a function h such that its error is greater than ϵ with a probability less than δ. Thus by choosing low values for ϵ and δ, we can have a low error (i.e. $< \epsilon$) with high probability (i.e. $1 - \delta$).

Next section clarifies the above idea with a practical example.

2.2 An Example of PAC Learnable Functions

Suppose that we need to learn the concept of medium-built people. Given such problem two very important features are the height and the weight of a person. One of these features alone would not be able to characterize the concept of medium-built body. For example, a person which has a height of 1,75 meters may be seen as medium person but if her/his weight is 130 kg we would immediately change our idea.

As the above two features assume real number values, we can represent people on a cartesian chart, where the X-axis and Y-axis correspond to height and the weight, respectively. Figure 3 illustrates such idea.

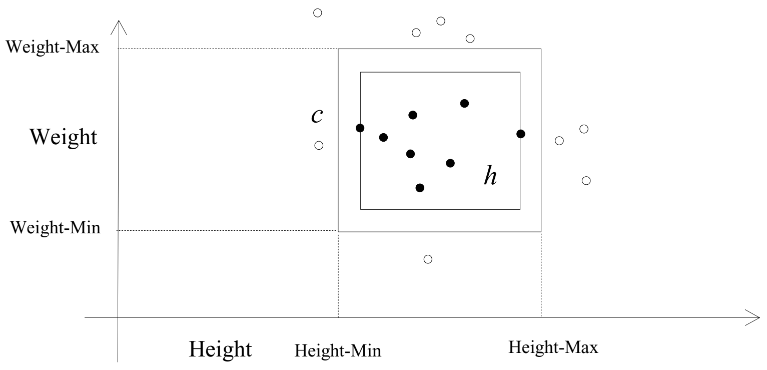


Fig. 3. The medium-built person concept on a cartesian chart

This representation implies that the medium-built person concept c is represented by a rectangle, which defines the maximum and minimum weight and height. Suppose that we have available some training examples, i.e. the measures of a set of people, which may or may not have a medium-build body, we can represent them in the chart. The white points, which are outside the rectangle c, are not medium-built people all the others (black points) are instead in such class.

As we assumed that our hypothesis c has a rectangular shape whose edges are parallel to the axes, our ML algorithm should only learn h from the rectangle set, namely the set of hypotheses H. Additionally, since the error is defined as $P[f(x) \neq h(x)]$, we can evaluate it by dividing the area between the rectangles c and h by the area of c[2].

In order to design an effective algorithm, we need to exploit the training data. In this respect, a simple way is to avoid errors in the training set; hence our learning algorithm is the following:

Select the smallest rectangle having its edges parallel to the axes that includes all training examples corresponding to medium-built people.

Since it includes all positive points, it would not make mistakes on them on the training set. Selecting also the smallest rectangle also prevents to commit error on the more external negative points.

We would like to verify that this is a PAC algorithm. To do this, we fix an error ϵ, a target probability δ and evaluate the $P[error(h) > \epsilon]$, i.e. the probability of generating a *bad* hypothesis, h. (to be a PAC algorithm, such probability must be lower then δ). Since $P[error(h) > \epsilon]$, h correctly classifies one training example with a probability $< 1 - \epsilon$. This implies that, in the cartesian representation of Figure 4.A, the rectangle associated with a *bad* h is included in the smallest rectangle of *good* hypotheses (i.e. the hypotheses of area equal to $1 - \epsilon$). Additionally, our algorithm produces a rectangle (a hypothesis) that includes all m training points.

Now, let us consider the four strips between c and h: a *bad* hypothesis cannot contemporary touch all four strips as shown by the frames B and C. It follows that, a

[2] It can be proven that this is true for any distribution D.

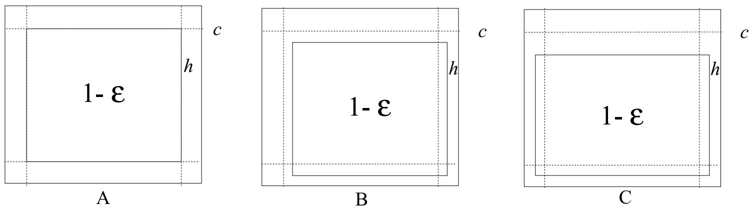


Fig. 4. Probabilities of *bad* and *good* hypotheses

necessary condition for the existence of a *bad* hypothesis is to have all the m points at least outside of one of the 4 strips. *Necessary* means that it must happen each time we learn a bad hypothesis and, consequently, the probability of drawing m points out of at least one strip is higher than a hypothesis to be *bad*. In other words, the latter is upper bounded by the former probability. More in detail, the evaluation of the probability of the latter follows these steps:

1. the probability that a point x is out of one strip, $P(x \text{ out of 1 strip}) = (1 - \epsilon/4)$;
2. the probability that m points are out of one strip, $P(x \text{ out of 1 strip})^m = (1-\epsilon/4)^m$;
3. the probability that m points are out of 4 strips $< 4P(x \text{ out of 1 strip})^m = 4(1 - \epsilon/4)^m$;

Therefore, we can use the inequality, $P[error(h) > \epsilon] < 4(1 - \epsilon/4)^m < \delta$, to impose our δ requirement. From $\Rightarrow 4(1 - \epsilon/4)^m < \delta$, we can derive an upperbound[3] to m (satisfying our constraint):

$$m > \frac{ln(\delta/4)}{ln(1 - \epsilon/4)}$$

From Taylor's series, we know that

$$-ln(1 - y) = y + y^2/2 + y^3/3 + .. \Rightarrow (1 - y) < e^{(-y)}$$

We can apply the above inequality to $ln(1 - \epsilon/4)$ to obtain

$$m > \frac{ln(\delta/4)}{ln(1 - \epsilon/4)} \Rightarrow m > \frac{4ln(4/\delta)}{\epsilon}. \tag{5}$$

Eq. 5 proves that the medium-built people concept is PAC learnable as we can reduce the error probability as much as we want, provided that we have an enough number of training examples.

It is interesting to note that a general upperbound for PAC functions can be evaluated by considering the following points:

1. the probability that a *bad* hypothesis is consistent with m training examples (i.e. classifies them correctly) is $(1 - \epsilon)^m$;

[3] Consider that we divide by $ln(1 - \epsilon/4)$, which is always negative, thus we need to change the direction of the inequality.

2. the number of *bad* hypotheses is less than the total number of hypotheses $N \Rightarrow$
3. $P(h$ bad and consistent with m examples) = $N(1-\epsilon)^m < Ne^{-\epsilon^m} = Ne^{-m\epsilon} < \delta$. It follows that
$$m > \frac{1}{\epsilon}(ln\frac{1}{\delta} + lnN). \tag{6}$$

We can use Eq. 6 when N can be estimated. For example, if we want to learn a Boolean function of n variable, their number is $2^{2^n} > N \Rightarrow$ a rough upperboud of the needed m is $\frac{1}{\epsilon}(ln\frac{1}{\delta} + 2^n ln2)$.

In most cases the above bound is not useful and we need to derive one specific to our target problem as we did for the medium-built concept. However, when the feature space is larger than 2 the manual procedure may become much more complex. In the next section, we will see a characterization of PAC functions via VC dimension, which makes more systematic derivation of PAC properties.

2.3 The VC-Dimension

The previous section has shown that some function classes can be learned with any accuracy and this depends on the properties of the adopted learning algorithm. For example, the fact that we use rectangles as our hypothesis space (the one from which our algorithm selects h) instead of circles or lines impacts on the *learning capacity* of our algorithm.

Indeed, it is easy to show that using lines, we would have never been able to separate medium-built people from the others whereas the rectangle class is rather effective to do this. Thus, we need a property that allows us to determine which hypothesis class is more appropriate to learn a target function $f \in F$. Moreover, we note that, in most of the cases, we do not know the nature of the target f. We know only the training examples, consequently, our property should be derived only from them and by the function class H that we have available.

The Vapnik and Chervonenkis (VC) dimension aims to characterize functions from a learning perspective. The intuitive idea is that different function classes have different capacity in separating data points: some of them can just separate some configurations of points whereas others can separate a much larger number of configurations, i.e. they are in some sense more general purpose. The VC dimension captures this kind of property.

Intuitively, VC dimension, i.e. the learning capacity, determines the generalization reachable during learning:

– A function selected from a high class capacity is expected to *easily* separate the training points since it has the capacity to adapt to any training set. This will result on a learned function too specific to the used training data (i.e. it will overfit data). An immediate consequence is that the probability to correctly separate the test set will be lower.
– In contrast, a function that belongs to a low capacity class can separate a lower number of data configurations thus if it successful separates the current training points, the probability to well separate the test data will be higher.

The definition of VC dimension depends on the concept of shattering a set of points.

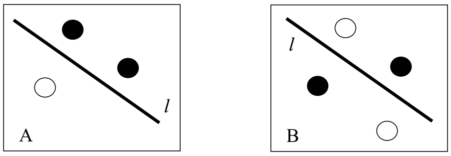


Fig. 5. VC dimension of lines in a bidimensional space

Definition 1. ***Shattered Sets***
Let us consider binary classification functions $f \in F$, $f : X \to \{0, 1\}$. We say that $S \subseteq X$ is shattered by a function class F if $\forall S' \subseteq S$, $\exists f \in F$:

$$f(x) = \begin{cases} 0 \textit{ iff } x \in S' \\ 1 \textit{ iff } x \in S - S' \end{cases} \tag{7}$$

The definition says that a set of points S is shattered by a function class F if for any assignment of the points in S into $\{0, 1\}$, we can find $f \in F$ that reproduces such assignments.

A graphical interpretation is given in Figure 5. In the frame A, we have 3 points represented in a two-dimensional space. The target function class L is the one of linear functions. For any assignment of points (white is 0 and black is 1), we can find a line $l \in L$ that separates them. From l we can derive the shattering function f by assigning $f(x_1, x_2)$=0 *iff* $x_2 < l(x_1)$ and 1 otherwise, i.e., if the point is under the line, we assign 0 to it and 1 otherwise. Consequently, a set of three points can be shattered by linear functions.

On the contrary, the 4 points in the frame B cannot be shattered. More precisely, there are not 4 points that can be shattered by linear functions since we can always draw a tetragon having such points as vertices and assign the same color to the opposite vertices. If the line assigned the same color to the opposite vertices there would always be a vertex on the same side of such two points with a different color.

Definition 2. ***VC dimension***
The VC dimension of a function class F is the maximum number of points that can be shattered by F.

Since Figure 5.A shows a set of tree points shattered by a linear function, such class has at least a VC dimension of 3 in the bidimensional space. We have also proved that 4 points cannot be shattered, consequently, the VC dimension of linear functions on the plane is exactly 3. Note that, selecting points that are linearly dependent, i.e., they lie on the same lines, will not work as we cannot hope to shatter them if we assign the same label to those external an a different color to the internal one. In particular it can be proven (see [16]) the following:

Theorem 1. *Consider a set of m points in $\mathbb{R}^n$ and choose any one of the points as origin, then they can be shattered by oriented hyperplanes if and only if the position vectors of the remaining points are linearly independent.*

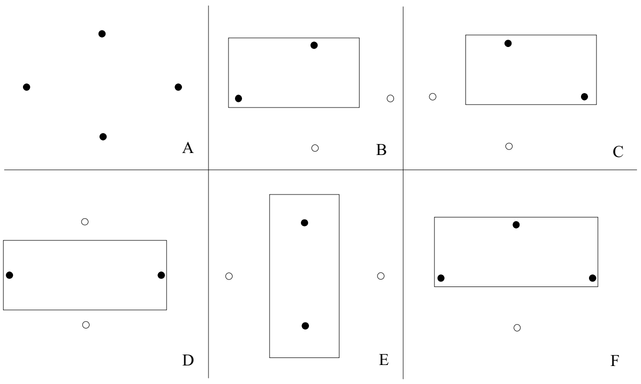


Fig. 6. VC dimension of (axis aligned) rectangles

As a consequence we have the following

Corollary 1. *The VC dimension of the class of functions composed by the set of oriented hyperplanes in $\mathbb{R}^n$ is n+1.*

Proof. We can always choose one of the points as origin of vectors and the remaining n points as their end such that the vectors are linearly independent. However, we can never choose $n + 1$ of such points (since no set of $n + 1$ vectors in $\mathbb{R}^n$ can be linearly independent).

This Corollary is useful to determine the VC dimension of linear functions in an n dimensional space. Linear functions are the building block of Support Vector Machines, nevertheless, there are other examples of classifiers which have different VC dimension such as the rectangle class. The following example is useful to understand how to evaluate the VC dimension of geometric classifiers.

Example 1. **The VC dimension of rectangles with edges parallel to the axes**
To evaluate the VC dimension of rectangles, we (i) make a guess about its value, for instance 4, (ii) show that 4 points can be shattered by rectangles and (iii) prove that no set of 5 points can be shattered.

Let us choose 4 points that are not aligned like in Figure 6.A. Then, we give all possible assignments to the 4 points. For example, Figure 6.B shows two pairs of adjacent points, which have the same color. In Section 2.2, we established that points inside the rectangle belong to medium-built people, i.e., they are positive examples of such class. Without loss of generality, we can keep such assumption and use the black color to indicate that the examples are positive (or that they are assigned to 1). The only relevant aspect is that we need to be consistent with such choice for all assignments, i.e., we cannot change our classification algorithm while we are testing it on the point configurations.

From the above convention, it follows that given the assignments B, C, D, E and F in Figure 6, we need to find the rectangles that contain only black points and leave the white points outside. The rectangles C, D, E, F separate half positive and half negative examples. It is worth noting that if we have 3 positive (or 3 negative) examples, finding the shattering rectangles is straight forward (see Frame E), consequently, we have proven that the VC dimension is at least 4.

To prove that is not greater than 4, let us consider a general 5 point set. We can create 4 different rankings of the points by sorting in ascending and descending order by their x-coordinate and by their y-coordinate. Then, we color the top point of each of the 4 lists in black and the 5th point in white. The latter will be included (by construction) in the rectangle of the selected 4 vertices. Since any rectangular hypothesis h that contains the four points must contain the previous rectangle, we cannot hope to exclude the 5th point from h. Consequently, no set of 5 points can be shattered by the rectangle class.

Finally, we report two theorems on the sample complexity, which, given a certain wished error, derive upper and lower bounds of the required number of training examples. We also report one theorem on the error probability of a hypothesis given the VC dimension of its class. These theorems make clear the link between VC dimension and PAC learning.

Theorem 2. *(upper bound on sample complexity, [15])*
Let H and F be two function classes such that $F \subseteq H$ and let A an algorithm that derives a function $h \in H$ consistent with m training examples. Then, $\exists c_0$ such that $\forall f \in F$, $\forall D$ distribution, $\forall \epsilon > 0$ and $\delta < 1$ if

$$m > \frac{c_0}{\epsilon}\Big(d \times ln\frac{1}{\epsilon} + \frac{1}{\delta}\Big)$$

then with a probability $1 - \delta$,

$$error_D(h) \leq \epsilon,$$

where d is the VC dimension of H and $error_D(h)$ is the error of h according to the data distribution D.

Theorem 3. *(lower bound on sample complexity, [15])*
To learn a concept class F whose VC-dimension is d, any PAC algorithm requires $m = O(\frac{1}{\epsilon}(\frac{1}{\delta} + d))$ examples.

Theorem 4. *(Vapnik and Chervonenkis, [64])*
Let H be a hypothesis space having VC dimension d. For any probability distribution D on $X \times \{-1, 1\}$, with probability $1 - \delta$ over m random examples S, any hypothesis $h \in H$ that is consistent with S has error no more than

$$error(h) \leq \epsilon(m, H, \delta) = \frac{2}{m}\big(d \times ln\frac{2e \times m}{d} + ln\frac{2}{\delta}\big),$$

provided that $d \leq m$ and $m \geq 2/\epsilon$.

Exercise 4. Compare the upper bounds on sample complexity of rectangles derived in Section 2.2 with the one derivable from Theorem 2.

Exercise 5. Evaluate the VC dimensions of triangles aligned and not aligned to the axes.

Exercise 6. Evaluate the VC dimension of circles.

3 Support Vector Machines

The previous section has shown that classification instances can be represented with numerical features. These can also be associated with the coordinates of points in an n-dimensional space, where a classification function can be modeled with geometrical objects, e.g., lines or hyperplanes. The latter constitute the basic building block of the statistical learning theory, which has produced Support Vector Machines (SVMs).

In this section, we first introduce the Perceptron algorithm, which can be considered the simplest SVM and then we define the theory and algorithms of more advanced SVMs. One of their important properties is the possibility to use kernel functions to deal with non linear classification problems. Thus, a conclusive section will introduce the kernel theory and its application to advanced learning tasks, e.g., the classification of syntactic-parse trees.

3.1 Perceptrons

Once objects are projected into a vector space, they can be simply classified by linear functions, e.g., Figure 5.A shows a line that separates black from white points. One advantage of such mathematical objects is their simplicity that allows us to design efficient learning algorithms, i.e., efficient approaches to find separating lines or hyperplanes in high dimensional spaces.

The reader may wonder if such simplicity limits the capability of the learning algorithms or if we can use them to learn any possible *learnable function*. It is clear that with only one hyperplane, we cannot learn any function. For example, Figure 5 shows four points that cannot be separated in the Frame B. However, this is not a definitive limitation of linear functions as:

1. By modeling our learning problem more effectively, i.e., by choosing more appropriate features, the target problem could become linearly separable. For example, the four points of the previous figure can be divided in a three-dimensional space. This means that we need just to add a significant feature to solve the problem.
2. We can use linear functions in cascade. The resulting function is more expressive and, depending on the number of levels in such cascade, we can design any function.

The thesis that linear functions are sufficient to derive any learnable relation from examples is supported by the observation that human beings' brain is structured with such sort of devices.

To clarify this point, let us consider an animal neuron shown in Figure 7. It is constituted by one set of inputs, i.e., the dendrites, which are connected to a cellular body, i.e., soma, via synapses. These are able to amplify or attenuate an incoming signal. The neuron output is transported by the axon, whose filaments are connected to the dendrites of other neurons. When a chemical signal is transmitted to the dendrites, it is amplified by the synapses before entering in the soma. If the overall signal, coming from different synapses, overcomes a certain threshold, the soma will launch a signal to the other neurons through the axon.

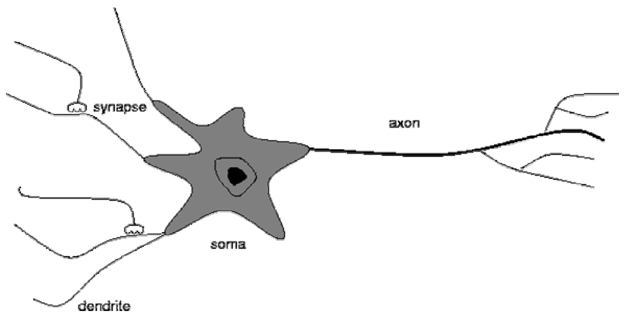


Fig. 7. An animal neuron

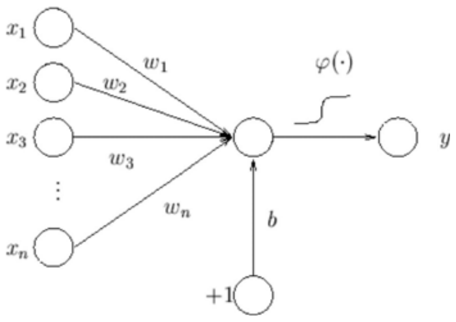


Fig. 8. An artificial neuron

The artificial version of the neuron is often referred to as *Perceptron* and can be sketched as in Figure 8. Each dendrite is an input x_i associated with a weight w_i. The product between the weights and the input signals are summed together and if such summation overcomes the threshold b the output y will be 1, otherwise it will be 0. The interesting aspect is that the output of such neuron can be modeled with a simple hyperplane whose equation is:

$$y = w_1 x_1 + .. + w_n x_n + b = \boldsymbol{w} \cdot \boldsymbol{x} + b = 0 \tag{8}$$

where the final perceptron classification function output is obtained by applying the signum function to y, i.e.,

$$f(\boldsymbol{x}) = sgn(\boldsymbol{w} \cdot \boldsymbol{x} + b) \tag{9}$$

Eq. 9 shows that linear functions are equivalent to neurons, which, combined together, constitute the most complex learning device that we know, i.e., the human brain. The signum function simply divides the data points in two sets: those that are over and those that are below the hyperplane. The major advantage of using linear functions is that given a set of training points, $\{\boldsymbol{x}_1, .., \boldsymbol{x}_m\}$, each one associated with a classification label y_i (i.e., $+1$ or -1), we can apply a learning algorithm that derives the vector $\boldsymbol{w}$ and the scalar b of a separating hyperplane, provided that at least one exists.

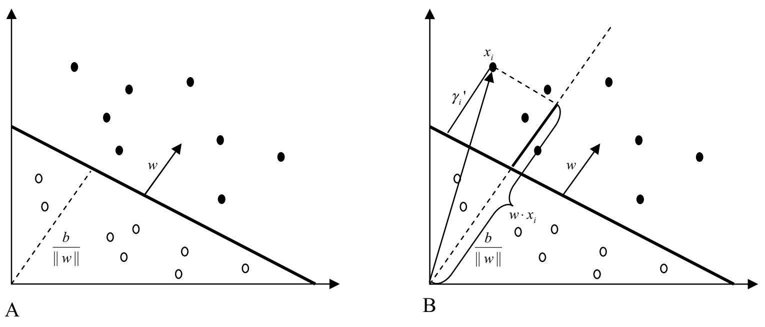


Fig. 9. Separating hyperplane and geometric margin

For example, Figure 9.A shows a set of training points (black positives and white negatives) along with a separating hyperplane in a 2-dimensional space. The vector $\boldsymbol{w}$ and the scalar $-b/||\boldsymbol{w}||$ are the gradient vector and the distance of such hyperplane from the origin, respectively. Indeed, from Eq. 8, $-b = \boldsymbol{w} \cdot \boldsymbol{x}$ thus $-b/||\boldsymbol{w}|| = \boldsymbol{w}/||\boldsymbol{w}|| \cdot \boldsymbol{x}$, where $\boldsymbol{x}$ is any point lying on the hyperplane and $\boldsymbol{w}/||\boldsymbol{w}|| \cdot \boldsymbol{x}$ is the projection of $\boldsymbol{x}$ on the gradient (i.e., the normal to the hyperplane).

The perceptron learning algorithm exploits the above properties along with the concept of functional and geometric margin.

Definition 3. *The **functional margin** γ_i of an example $\boldsymbol{x}_i$ with respect to a hyperplane $\boldsymbol{w} \cdot \boldsymbol{x} + b = 0$ is the product $y_i(\boldsymbol{w} \cdot \boldsymbol{x}_i + b)$.*

Definition 4. *The **geometric margin** γ_i' of an example $\boldsymbol{x}_i$ with respect to a hyperplane $\boldsymbol{w} \cdot \boldsymbol{x} + b = 0$ is $y_i(\frac{\boldsymbol{w}}{||\boldsymbol{w}||} \cdot \boldsymbol{x}_i + \frac{b}{||\boldsymbol{w}||})$.*

It is immediate to see in Figure 9.B that the geometric margin γ_i' is the distance of the point $\boldsymbol{x}_i$ from the hyperplane as:

- $\frac{\boldsymbol{w}}{||\boldsymbol{w}||} \cdot \boldsymbol{x}_i$ is the projection of $\boldsymbol{x}_i$ on the line crossing the origin and parallel to $\boldsymbol{w}$;
- the distance of the hyperplane from the origin is subtracted to the above quantity, i.e., $\frac{b}{||\boldsymbol{w}||}$. It follows that we obtain the distance of $\boldsymbol{x}$ from the hyperplane.
- When the example $\boldsymbol{x}$ is negative, it is located under the hyperplane thus the product $\boldsymbol{w} \cdot \boldsymbol{x}_i$ is negative. If we multiply such quantity by the label y_i (i.e., -1), we make it positive, i.e., we obtain a distance.

Given the above geometric concepts, the algorithm of perceptron learning in Table 2, results very clear. At step $k = 0$, $\boldsymbol{w}$ and b are set to 0, i.e., $\boldsymbol{w}_0 = \boldsymbol{0}$ and $b_0 = 0$, whereas R is set to the maximum length of the training set vectors, i.e., the maximum among the distances of the training points from the origin. Then, for each $\boldsymbol{x}_i$, the functional margin $y_i(\boldsymbol{w}_k \cdot \boldsymbol{x}_i + b_k)$ is evaluated. If it is negative it means that $\boldsymbol{x}_i$ is not correctly classified by the hyperplane, i.e., y_i disagrees with the point position with respect to the hyperplane. In this case, we need to adjust the hyperplane to correctly classify the example. This can be done by rotating the current hyperplane (i.e., by summing $\eta y_i \boldsymbol{x}_i$

Table 2. Rosenblatt's perceptron algorithm

```
function Perceptron(training-point set: ${x_1, .., x_m}$)
begin
  $w_0 = 0$; $b_0 = 0$; $k = 0$;
  $R = \max_{1 \le i \le m} ||x_i||$
  repeat
    no_errors = 1;
    for ($i = 1$ to $m$)
      if $y_i(w_k \cdot x_i + b_k) \le 0$ then
        $w_{k+1} = w_k + \eta y_i x_i$;
        $b_{k+1} = b_k + \eta y_i R^2$;
        $k = k + 1$;no_errors = 0;
      end(if)
  until no_errors;
  return $k$, $w_k$ and $b_k$ ;
end
```

to $\boldsymbol{w}_k$) as shown in the charts A and B of Figure 10 and by translating the hyperplane of a quantity $\eta y_i R^2$ as shown in the chart C.

The perceptron algorithm always converges when the data points are linearly separable as stated by the following.

Theorem 5. *(Novikoff) Let S be a non-trivial training and let $\gamma > 0$ and $R = max_{1 \le i \le m} ||\boldsymbol{x}_i||$. Suppose that there exists a vector $\boldsymbol{w}_{opt}$ such that $||\boldsymbol{w}_{opt}|| = 1$ and $y_i(\boldsymbol{w}_{opt} \cdot \boldsymbol{x}_i + b_{opt}) \ge \gamma \ \forall i = 1, .., m$. Then the number of mistakes made by the perceptron algorithm on S is at most $\left(\frac{2R}{\gamma}\right)^2$.*

This theorem proves that the algorithm converges in a finite number of iterations bounded by $\left(\frac{2R}{\gamma}\right)^2$ provided that a separating hyperplane exists. In particular:

- the condition $||\boldsymbol{w}_{opt}|| = 1$ states that normalized vectors are considered, i.e. $\boldsymbol{w}_{opt} = \frac{\boldsymbol{w}_{opt}}{||\boldsymbol{w}_{opt}||}$, thus the functional margin is equal to the geometric margin.
- $y_i(\boldsymbol{w}_{opt} \cdot \boldsymbol{x}_i + b_{opt}) \ge \gamma$ is equivalent to state that for such hyperplane the geometric margin of the data points are $\ge \gamma > 0$, i.e. any point is correctly classified by the hyperplane, $\boldsymbol{w}_{opt} \cdot \boldsymbol{x} + b_{opt} = 0$.

If the training data is not separable then the algorithm will oscillate indefinitely correcting at each step some misclassified example.

An interesting property showed by the Novikoff theorem is that the gradient $\boldsymbol{w}$ is obtained by adding vectors proportional to the examples $\boldsymbol{x}_i$ to $\boldsymbol{0}$. This means that $\boldsymbol{w}$ can be written as a linear combination of training points, i.e.,

$$\boldsymbol{w} = \sum_{i=1}^{m} \alpha_i y_i \boldsymbol{x}_i \tag{10}$$

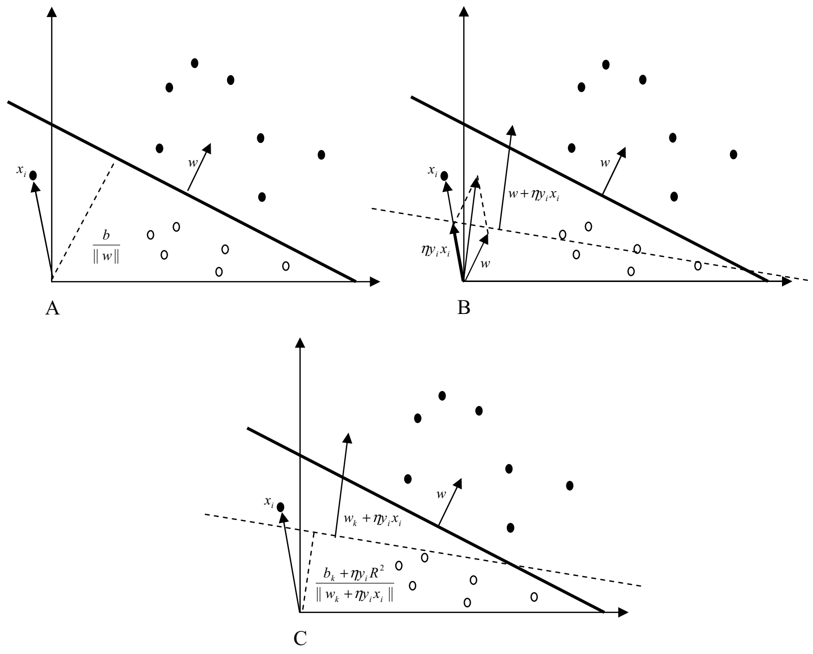


Fig. 10. Perceptron algorithm process

Since the sign of the contribution $\boldsymbol{x}_i$ is given by y_i, α_i is positive and is proportional (through the η factor) to the number of times that $\boldsymbol{x}_i$ is incorrectly classified. *Difficult points* that cause many mistakes will be associated with large α_i.

It is interesting to note that, if we fix the training set S, we can use the α_i as alternative coordinates of a dual space to represent the target hypothesis associated with $\boldsymbol{w}$. The resulting decision function is the following:

$$h(x) = sgn(\boldsymbol{w} \cdot \boldsymbol{x} + b) = sgn\left(\Big(\sum_{i=1}^{m} \alpha_i y_i \boldsymbol{x}_i\Big) \cdot \boldsymbol{x} + b\right) =$$

$$= sgn\left(\sum_{i=1}^{m} \alpha_i y_i (\boldsymbol{x}_i \cdot \boldsymbol{x}) + b\right) \tag{11}$$

Given the dual representation, we can adopt a learning algorithm that works in the dual space described in Table 3.

Note that as the Novikoff's theorem states that the learning rate η only changes the scaling of the hyperplanes, it does not affect the algorithm thus we can set $\eta = 1$. On the contrary, if the perceptron algorithm starts with a different initialization, it will find a different separating hyperplane. The reader may wonder if such hyperplanes are all equivalent in terms of the classification accuracy of the test set; the answer is no: different hyperplanes may lead to different error probabilities. In particular, the next

Table 3. Dual perceptron algorithm

```
function Perceptron(training-point set: ${\boldsymbol{x}_1, .., \boldsymbol{x}_m}$)
begin
    $\boldsymbol{\alpha} = \mathbf{0}$; $b_0 = 0$;
    $R = \max_{1 \le i \le m} ||\boldsymbol{x}_i||$
    repeat
        no_errors = 1;
        for ($i = 1$ to $m$)
            if $y_i\left(\sum_{j=1}^{m} \alpha_j y_j (\boldsymbol{x}_j \cdot \boldsymbol{x}) + b\right) \le 0$ then
                $\alpha_i = \alpha_i + 1$;
                $b = b + y_i R^2$;
                no_errors = 0;
            end(if)
    until no_errors;
    return $\boldsymbol{\alpha}$ and $b$ ;
end
```

section shows that the maximal margin hyperplane minimizes an upperbound to the error probability on the space of all possible hyperplanes.

3.2 Maximal Margin Classifier

The PAC theory suggests that, for a class of target functions, a hypothesis h that is learned consistently with the training set provides low error probability and we can show an analytical bound for such error. This idea can be applied to hyperplanes to estimate the final error probability but also to improve the learning algorithm of linear classifiers. Indeed, one of the interesting results of the statistical learning theory is that to reduce such probability, we need to select the hyperplane (from the set of separating hyperplanes) that shows the maximum distance between positive and negative examples. To understand better this idea let us introduce some definitions:

Definition 5. *The* ***functional (geometric) margin distribution*** *of a hyperplane ($\boldsymbol{w}$,b) with respect to a training set S is the distribution of the functional (geometric) margins of the examples, i.e. $y_i(\boldsymbol{w} \cdot \boldsymbol{x}_i + b) \forall \boldsymbol{x}_i \in S$.*

Definition 6. *The* ***functional (geometric) margin of a hyperplane*** *is the minimum functional (geometric) margin of the distribution.*

Definition 7. *The* ***functional (geometric) margin of a training set*** *S is the maximum functional (geometric) margin of a hyperplane over all possible hyperplanes. The hyperplane that realizes such maximum is called the* ***maximal margin hyperplane****.*

Figure 11 shows the geometric margins of the points $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ (part A) and the geometric margin of the hyperplane (part B) whereas Figure 12 shows two separating hyperplanes that realize two different margins.

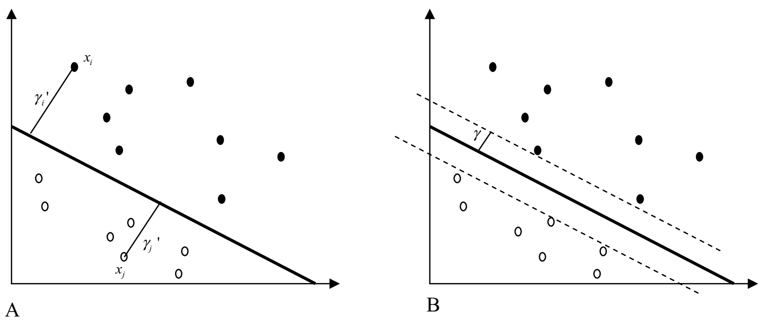


Fig. 11. Geometric margins of two points (part A) and margin of the hyperplane (part B)

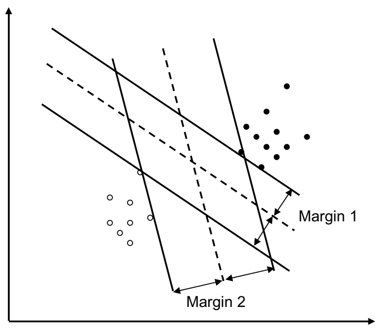


Fig. 12. Margins of two hyperplanes

Intuitively, the larger the margin of a hyperplane is, the lower the probability of error is. The important result of the statistical learning theory is that (i) analytical upperbounds to such error can be found; (ii) they can be proved to be correlated to the hyperplanes; and (iii) the maximal margin hyperplane is associated with the lowest bound.

In order to show such analytical result let us focus on finding the maximal margin hyperplane. Figure 13 shows that a necessary and sufficient condition for a hyperplane $\boldsymbol{w} \cdot \boldsymbol{x} + b = 0$ to be a maximal margin hyperplane is that (a) two *frontier* hyperplanes (negative and positive frontiers) exist and (b) they hold the following properties:

1. their equations are $\boldsymbol{w} \cdot \boldsymbol{x} + b = k$ and $\boldsymbol{w} \cdot \boldsymbol{x} + b = -k$, i.e. they are parallel to the target hyperplane and are both located at a distance of k ($\frac{k}{||\boldsymbol{w}||}$ if $\boldsymbol{w}$ is not a normalized vector;
2. such equations satisfy the constraints $y_i(\boldsymbol{w} \cdot \boldsymbol{x}_i + b) \geq k \; \forall x_i \in S$, i.e. they both separate the data points in S; and
3. the distance of the hyperplane from such frontiers is maximal with respect to other frontiers.

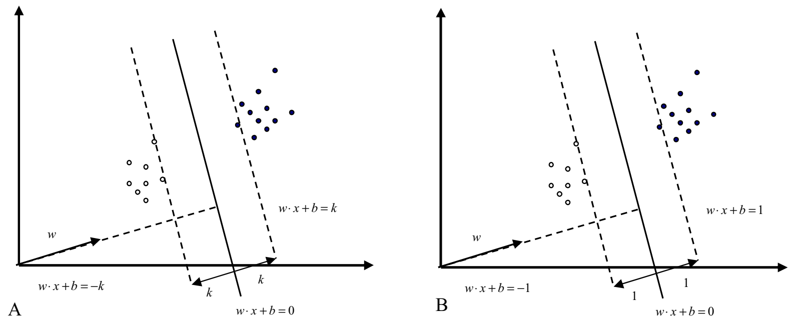


Fig. 13. Frontiers and Maximal Margin Hyperplane

First, property 1 follows from a simple consideration: suppose that: (i) the nearest positive example $\boldsymbol{x}^+$ is located at a distance of γ_i from a hyperplane h_1; (ii) the nearest negative example $\boldsymbol{x}^-$ is located at a distance of γ_j ($\neq \gamma_i$); and (iii) the h_1 margin is the minimum between γ_i and γ_j. If we select a hyperplane h_2 parallel to h_1 and equidistant from $\boldsymbol{x}^+$ and $\boldsymbol{x}^-$, it will be at a distance of $k = \frac{\gamma_i + \gamma_j}{2}$ from both $\boldsymbol{x}^+$ and $\boldsymbol{x}^-$. Since $k \geq min\{\gamma_i, \gamma_j\}$, the margin of h_2 equidistant from the frontier points is always greater or equal than other hyperplanes.

Second, the previous property has shown that the nearest positive examples is located on the frontier $\boldsymbol{w} \cdot \boldsymbol{x} + b = k$ thus all the other positive examples $\boldsymbol{x}^+$ have a functional margin $\boldsymbol{w} \cdot \boldsymbol{x}^+ + b$ larger than k. The same rational applies to the negative examples but, to work with positive quantities, we multiply $(\boldsymbol{w} \cdot \boldsymbol{x}_i + b)$ by the label y_i, thus, we obtain the constrain $y_i(\boldsymbol{w} \cdot \boldsymbol{x}_i + b_k) \geq k$.

Finally, the third property holds since $\frac{k}{||\boldsymbol{w}||}$ is the distance from one of the two frontier hyperplanes which, in turn, is the distance from the nearest points, i.e. the margin.

From these properties, it follows that the maximal margin hyperplane can be derived by solving the optimization (maximization) problem below:

$$\begin{cases} max & \frac{k}{||\boldsymbol{w}||} \\ y_i(\boldsymbol{w} \cdot \boldsymbol{x_i} + b) \geq 1 & \forall \boldsymbol{x}_i \in S, \end{cases} \tag{12}$$

where $\frac{k}{||\boldsymbol{w}||}$ is the objective function, $y_i(\boldsymbol{w} \cdot \boldsymbol{x_i} + b) = 1 \quad \forall \boldsymbol{x}_i \in S$ are the set of linear equality constraints $h_i(\boldsymbol{w})$ and $y_i(\boldsymbol{w} \cdot \boldsymbol{x_i} + b) > 1 \quad \forall \boldsymbol{x}_i \in S$ are the set of linear inequality constraints, $g_i(\boldsymbol{w})$. Note that (i) the objective function is quadratic since $||\boldsymbol{w}|| = \boldsymbol{w} \cdot \boldsymbol{w}$ and (ii) we can rescale the distance among the data points such that the maximal margin hyperplane has a margin of exactly 1. Thus, we can rewrite Eq. 12 as follows:

$$\begin{cases} max & \frac{1}{||\boldsymbol{w}||} \\ y_i(\boldsymbol{w} \cdot \boldsymbol{x_i} + b) \geq 1 & \forall \boldsymbol{x}_i \in S \end{cases} \tag{13}$$

Moreover, we can transform the above maximization problem in the following minimization problem:

$$\begin{cases} min \quad ||\boldsymbol{w}|| \\ y_i(\boldsymbol{w} \cdot \boldsymbol{x_i} + b) \geq 1 \quad \forall \boldsymbol{x}_i \in S \end{cases} \tag{14}$$

Eq. 14 states that to obtain a maximal margin hyperplane, we have to minimize the norm of the gradient $\boldsymbol{w}$ but it does not provide any analytical evidence on the benefit of choosing such hyperplane. In contrast, the PAC learning theory provides the link with the error probability with the following theorem:

Theorem 6. *(Vapnik, 1982) Consider hyperplanes $\boldsymbol{w} \cdot \boldsymbol{x} + b = 0$ in a $\mathbb{R}^n$ vector space as hypotheses. If all examples $\boldsymbol{x_i}$ are contained in a ball of radius R and*

$$\forall \boldsymbol{x}_i \in S, \quad y_i(\boldsymbol{w} \cdot \boldsymbol{x_i} + b) \geq 1, \quad with \quad ||\boldsymbol{w}|| \leq A$$

then this set of hyperplanes has a VC-dimension d bounded by

$$d \leq min(R^2 \times A^2, n) + 1$$

The theorem states that if we set our hypothesis class H_A to be the set of hyperplanes whose $\boldsymbol{w}$ has a norm $\leq A$ then the VC dimension is less or equal than $R^2 \times A^2$. This means that if we reduce $||\boldsymbol{w}||$, we obtain a lower A and consequently a lower VC dimension, which in turn is connected to the error probability by the Theorem 4 (lower VC dim. results in lower error bound). This proves that, when the number of training examples is fixed, a lower VC-dimension will produce a lower error probability. In other words, as the maximum margin hyperplane minimizes the bound on the error probability, it constitutes a promising hypothesis for our learning problem.

Other interesting properties of the maximum margin hyperplane are derived from the optimization theory of convex functions over linear constraints. The main concepts of such theory relate on the following definition and theorem:

Definition 8. *Given an optimization problem with objective function $f(\boldsymbol{w})$, and equality constraints $h_i(\boldsymbol{w}) = 0$, $i = 1, .., l$, we define the Lagrangian function as*

$$L(\boldsymbol{w}, \boldsymbol{\beta}) = f(\boldsymbol{w}) + \sum_{i=1}^{l} \beta_i h_i(\boldsymbol{w}),$$

where the coefficient β_i are called Lagrange multipliers.

Theorem 7. *(Lagrange) A necessary condition for a normal point $\boldsymbol{w}^*$ to be a minimum of $f(w)$ subject to $h_i(\boldsymbol{w}) = 0$, $i = 1, .., l$, with f, $h_i \in C$ is*

$$\frac{\partial L(\boldsymbol{w}^*, \boldsymbol{\beta}^*)}{\partial \boldsymbol{w}} = \mathbf{0} \tag{15}$$

$$\frac{\partial L(\boldsymbol{w}^*, \boldsymbol{\beta}^*)}{\partial \boldsymbol{\beta}} = \mathbf{0} \tag{16}$$

for some values of $\boldsymbol{\beta}^*$. The above conditions are also sufficient provided that $\partial L(\boldsymbol{\beta}^*)$ is a convex function of $\boldsymbol{w}$.

Proof. (necessity)
A continue function has a local maximum (minimum) when the partial derivatives are equal 0, i.e. $\frac{\partial f(\boldsymbol{w})}{\partial \boldsymbol{w}} = \mathbf{0}$. Since, we are in presence of constraints, it is possible that $\frac{\partial f(\boldsymbol{w}^*)}{\partial \boldsymbol{w}} \neq \mathbf{0}$. To respect such equality constraints, given the starting point $\boldsymbol{w}^*$, we can move only perpendicularly to $\frac{\partial h_i(\boldsymbol{w}^*)}{\partial \boldsymbol{w}}$. In other words, we can only move perpendicularly to the subspace V spanned by the vectors $\frac{\partial h_i(\boldsymbol{w}^*)}{\partial \boldsymbol{w}}, i = 1,..,l$. Thus, if a point $\frac{\partial f(\boldsymbol{w}^*)}{\partial \boldsymbol{w}}$ lies on V, any direction we move causes to violate the constraints. In other words, if we start from such point, we cannot increase the objective function, i.e. it can be a minimum or maximum point. The V memberships can be stated as the linear dependence between $\frac{\partial f(\boldsymbol{w}^*)}{\partial \boldsymbol{w}}$ and $\frac{\partial h_i(\boldsymbol{w}^*)}{\partial \boldsymbol{w}}$, formalized by the following equation:

$$\frac{\partial f(\boldsymbol{w}^*)}{\partial \boldsymbol{w}} + \sum_{i=1}^{l} \beta_i \frac{\partial h_i(\boldsymbol{w}^*)}{\partial \boldsymbol{w}} = \mathbf{0} \tag{17}$$

where $\exists i : \beta_i \neq 0$. This is exactly the condition 15. Moreover, Condition 16 holds since $\frac{\partial L(\boldsymbol{w}^*,\boldsymbol{\beta}^*)}{\partial \boldsymbol{\beta}} = (h_1(\boldsymbol{w}^*), h_2(\boldsymbol{w}^*), ..., h_l(\boldsymbol{w}^*))$ and all the constraints $h_i(\boldsymbol{w}^*) = 0$ are satisfied for the feasible solution $\boldsymbol{w}^*$. $\square$

The above conditions can be applied to evaluate the maximal margin classifier, i.e. the Problem 14, but the general approach is to transform Problem 14 in an equivalent problem, simpler to solve. The output of such transformation is called dual problem and it is described by the following definition.

Definition 9. *Let $f(\boldsymbol{w})$, $h_i(\boldsymbol{w})$ and $g_i(\boldsymbol{w})$ be the objective function, the equality constraints and the inequality constraints (i.e. $\leq$) of an optimization problem, and let $L(\boldsymbol{w}, \boldsymbol{\alpha}, \boldsymbol{\beta})$ be its Lagrangian, defined as follows:*

$$L(\boldsymbol{w}, \boldsymbol{\alpha}, \boldsymbol{\beta}) = f(\boldsymbol{w}) + \sum_{i=1}^{m} \alpha_i g_i(\boldsymbol{w}) + \sum_{i=1}^{l} \beta_i h_i(\boldsymbol{w})$$

The ***Lagrangian dual problem*** *of the above primal problem is*

$$maximize \quad \theta(\boldsymbol{\alpha}, \boldsymbol{\beta})$$

$$subject\ to \quad \boldsymbol{\alpha} \geq \mathbf{0}$$

where $\theta(\boldsymbol{\alpha}, \boldsymbol{\beta}) = inf_{w \in W} \ \ L(\boldsymbol{w}, \boldsymbol{\alpha}, \boldsymbol{\beta})$

The *strong duality theorem* assures that an optimal solution of the dual is also the optimal solution for the primal problem and vice versa, thus, we can focus on the transformation of Problem 14 according to the Definition 9.

First, we observe that the only constraints in Problem 14 are the inequalities[4] $[g_i(\boldsymbol{w})$ = $-(y_i(\boldsymbol{w} \cdot \boldsymbol{x_i} + b) - 1)] \geq 0 \quad \forall \boldsymbol{x}_i \in S$.

[4] We need to change the sign of the inequalities to have them in the normal form, i.e. $g_i(\cdot) \leq 0$.

Second, the objective function is $\boldsymbol{w} \cdot \boldsymbol{w}$. Consequently, the primal Lagrangian[5] is

$$L(\boldsymbol{w}, b, \boldsymbol{\alpha}) = \frac{1}{2}\boldsymbol{w} \cdot \boldsymbol{w} - \sum_{i=1}^{m} \alpha_i [y_i(\boldsymbol{w} \cdot \boldsymbol{x_i} + b) - 1], \tag{18}$$

where α_i are the Lagrange multipliers and b is the extra variable associated with the threshold.

Third, to evaluate $\theta(\boldsymbol{\alpha}, \boldsymbol{\beta}) = inf_{w \in W} \;\; L(\boldsymbol{w}, \boldsymbol{\alpha}, \boldsymbol{\beta})$, we can find the minimum of the Lagrangian by setting the partial derivatives to 0.

$$\frac{\partial L(\boldsymbol{w}, b, \boldsymbol{\alpha})}{\partial \boldsymbol{w}} = \boldsymbol{w} - \sum_{i=1}^{m} y_i \alpha_i \boldsymbol{x}_i = \boldsymbol{0} \quad \Rightarrow \quad \boldsymbol{w} = \sum_{i=1}^{m} y_i \alpha_i \boldsymbol{x}_i \tag{19}$$

$$\frac{\partial L(\boldsymbol{w}, b, \boldsymbol{\alpha})}{\partial b} = \sum_{i=1}^{m} y_i \alpha_i = 0 \tag{20}$$

Finally, by substituting Eq. 19 and 20 into the primal Lagrangian we obtain

$$\begin{aligned} L(\boldsymbol{w}, b, \boldsymbol{\alpha}) &= \frac{1}{2}\boldsymbol{w} \cdot \boldsymbol{w} - \sum_{i=1}^{m} \alpha_i [y_i(\boldsymbol{w} \cdot \boldsymbol{x_i} + b) - 1] = \\ &= \frac{1}{2} \sum_{i,j=1}^{m} y_i y_j \alpha_i \alpha_j \boldsymbol{x_i} \cdot \boldsymbol{x_j} - \sum_{i,j=1}^{m} y_i y_j \alpha_i \alpha_j \boldsymbol{x_i} \cdot \boldsymbol{x_j} + \sum_{i=1}^{m} \alpha_i \\ &= \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{m} y_i y_j \alpha_i \alpha_j \boldsymbol{x_i} \cdot \boldsymbol{x_j} \end{aligned} \tag{21}$$

which according to the Definition 9 is the optimization function of the dual problem subject to $\alpha_i \geq 0$. In summary, the final dual optimization problem is the following:

$$\begin{aligned} &maximize \quad \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{m} y_i y_j \alpha_i \alpha_j \boldsymbol{x_i} \cdot \boldsymbol{x_j} \\ &subject\ to \quad \alpha_i \geq 0, \quad i = 1,..,m \\ &\qquad\qquad\quad \sum_{i=1}^{m} y_i \alpha_i = 0 \end{aligned}$$

where $\boldsymbol{w} = \sum_{i=1}^{m} y_i \alpha_i \boldsymbol{x}_i$ and $\sum_{i=1}^{m} y_i \alpha_i = 0$ are the relation derived from eqs. 19 and 20. Other conditions establishing interesting properties can be derived by the Khun-Tucker theorem. This provides the following relations for an optimal solution:

[5] As $\boldsymbol{w} \cdot \boldsymbol{w}$ or $\frac{1}{2}\boldsymbol{w} \cdot \boldsymbol{w}$ is the same optimization function from a solution perspective, we use the $\frac{1}{2}$ factor to simplify the next computation.

$$\begin{aligned}
\frac{\partial L(\boldsymbol{w}^*, \boldsymbol{\alpha}^*, \boldsymbol{\beta}^*)}{\partial \boldsymbol{w}} &= \mathbf{0} \\
\frac{\partial L(\boldsymbol{w}^*, \boldsymbol{\alpha}^*, \boldsymbol{\beta}^*)}{\partial \boldsymbol{\beta}} &= \mathbf{0} \\
\alpha_i^* g_i(\boldsymbol{w}^*) &= 0, \quad i = 1, .., m \\
g_i(\boldsymbol{w}^*) &\leq 0, \quad i = 1, .., m \\
\alpha_i^* &\geq 0, \quad i = 1, .., m
\end{aligned}$$

The third equation is usually called Karush-Khun-Tucker condition and it is very interesting for Support Vector Machines as it states that $\alpha_i^* \times [y_i(\boldsymbol{w} \cdot \boldsymbol{x_i} + b) - 1] = 0$. On one hand, if $\alpha_i^* = 0$ the training point $\boldsymbol{x}_i$ does not affect $\boldsymbol{w}$ as stated by Eq. 19. This means that the separating hyperplane and the associated classification function do not depend on such vectors. On the other hand, if $\alpha_i^* \neq 0 \Rightarrow [y_i(\boldsymbol{w} \cdot \boldsymbol{x_i} + b) - 1] = 0$ $\Rightarrow y_i(\boldsymbol{w} \cdot \boldsymbol{x_i} + b) = -1$, i.e. $\boldsymbol{x_i}$ is located on the frontier. Such data points are called support vectors (SV) as they support the classification function. Moreover, they can be used to derive the threshold b by evaluating the average between the projection of a positive and a negative SV on the gradient $\boldsymbol{w}^*$, i.e.:

$$b^* = -\frac{\boldsymbol{w}^* \cdot \boldsymbol{x}^+ + \boldsymbol{w}^* \cdot \boldsymbol{x}^-}{2}$$

The error probability upperbound of SVMs provides only a piece of evidence of the maximal margin effectiveness. Unfortunately, there is no analytical proof that such approach produces the *best* linear classifier. Indeed, it may exist other bounds lower than the one derived with the VC dimension and the related theory. Another drawback of the maximal margin approach is that it can only be applied when training data is linearly separable, i.e. the constraints over the negative and positive examples must be satisfied. Such *hard* conditions also define the name of such model, i.e., Hard Margin Support Vector Machines. In contrast, the next section introduces the Soft Margin Support Vector Machines, whose optimization problem relaxes some constraints, i.e., a certain number of errors on the training set is allowed.

3.3 Soft Margin Support Vector Machines

In real scenario applications, training data is often affected by noise due to several reasons, e.g. classification mistakes of annotators. These may cause the data not to be separable by any linear function. Additionally, the target problem itself may be not separable in the designed feature space. As result, the Hard Margin SVMs will fail to converge.

In order to solve such critical aspect, the Soft Margin SVMs have been designed. Their main idea is to allow the optimization problem to provide solutions that can violate a certain number of constraints. Intuitively, to be as much as possible consistent with the training data, such number of errors should be the lowest possible. This trade-off between the separability with highest margin and the number of errors can be encoded by (a) introducing *slack variables* ξ_i in the inequality constraints of Problem 14 and (b) the number of errors as quantity to be minimized in the objective function. The resulting optimization problem is

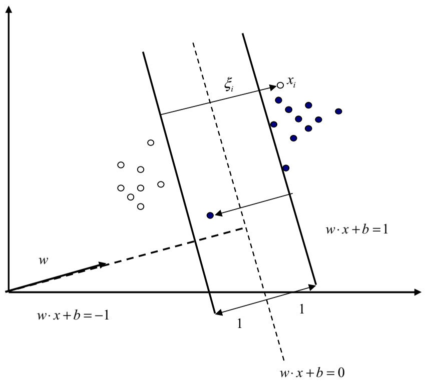


Fig. 14. Soft Margin Hyperplane

$$\begin{cases} min \quad ||\boldsymbol{w}|| + C\sum_{i=1}^{m} \xi_i^2 \\ y_i(\boldsymbol{w} \cdot \boldsymbol{x_i} + b) \geq 1 - \xi_i, \quad \forall i = 1,..,m \\ \xi_i \geq 0, \quad i = 1,..,m \end{cases} \tag{22}$$

whose the main characteristics are:

- The constraint $y_i(\boldsymbol{w} \cdot \boldsymbol{x_i} + b) \geq 1 - \xi_i$ allows the point $\boldsymbol{x}_i$ to violate the hard constraint of Problem 14 of a quantity equal to ξ_i. This is clearly shown by the outliers in Figure 14, e.g. $\boldsymbol{x}_i$.
- If a point is misclassified by the hyperplane then the slack variable assumes a value larger than 1. For example, Figure 14 shows the misclassified point x_i and its associated slack variable ξ_i, which is necessarily > 1. Thus, $\sum_{i=1}^{m} \xi_i$ is an upperbound to the number of errors. The same property is held by the quantity, $\sum_{i=1}^{m} \xi_i^2$, which can be used as an alternative bound[6].
- The constant C tunes the trade-off between the classification errors and the margin. The higher C is, the lower number of errors will be in the optimal solution. For $C \rightarrow \infty$, Problem 22 approximates Problem 14.
- Similarly to the hard margin formulation, it can be proven that minimizing $||\boldsymbol{w}|| + C\sum_{i=1}^{m} \xi_i^2$ minimizes the error probability of classifiers. Even though these are not perfectly consistent with the training data (they do not necessarily classify correctly all the training data).
- Figure 15 shows that by accepting some errors, it is possible to find better hypotheses. In the part A, the point $\boldsymbol{x}_i$ prevents to derive a good margin. As we accept to mistake $\boldsymbol{x}_i$, the learning algorithm can find a more suitable margin (part B).

As it has been done for the hard optimization problem, we can evaluate the primal Lagrangian:

$$L(\boldsymbol{w}, b, \boldsymbol{\xi}, \boldsymbol{\alpha}) = \frac{1}{2}\boldsymbol{w} \cdot \boldsymbol{w} + \frac{C}{2}\sum_{i=1}^{m} \xi_i^2 - \sum_{i=1}^{m} \alpha_i [y_i(\boldsymbol{w} \cdot \boldsymbol{x_i} + b) - 1 + \xi_i], \tag{23}$$

[6] This also results in an easier mathematical solution of the optimization problem.

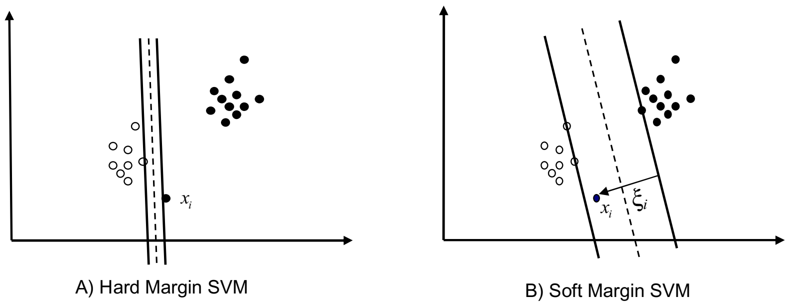


Fig. 15. Soft Margin vs. Hard Margin hyperplanes

where α_i are Lagrangian multipliers.

The dual problem is obtained by imposing stationarity on the derivatives respect to $\boldsymbol{w}$, $\boldsymbol{\xi}$ and b:

$$\begin{aligned}
\frac{\partial L(\boldsymbol{w}, b, \boldsymbol{\xi}, \boldsymbol{\alpha})}{\partial \boldsymbol{w}} &= \boldsymbol{w} - \sum_{i=1}^{m} y_i \alpha_i \boldsymbol{x}_i = \mathbf{0} \quad \Rightarrow \quad \boldsymbol{w} = \sum_{i=1}^{m} y_i \alpha_i \boldsymbol{x}_i \\
\frac{\partial L(\boldsymbol{w}, b, \boldsymbol{\xi}, \boldsymbol{\alpha})}{\partial \boldsymbol{\xi}} &= C\boldsymbol{\xi} - \boldsymbol{\alpha} = \mathbf{0} \\
\frac{\partial L(\boldsymbol{w}, b, \boldsymbol{\xi}, \boldsymbol{\alpha})}{\partial b} &= \sum_{i=1}^{m} y_i \alpha_i = 0
\end{aligned} \tag{24}$$

By substituting the above relations into the primal, we obtain the following dual objective function:

$$\begin{aligned}
w(\boldsymbol{\alpha}) &= \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{m} y_i y_j \alpha_i \alpha_j \boldsymbol{x_i} \cdot \boldsymbol{x_j} + \frac{1}{2C} \boldsymbol{\alpha} \cdot \boldsymbol{\alpha} - \frac{1}{C} \boldsymbol{\alpha} \cdot \boldsymbol{\alpha} = \\
&= \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{m} y_i y_j \alpha_i \alpha_j \boldsymbol{x_i} \cdot \boldsymbol{x_j} - \frac{1}{2C} \boldsymbol{\alpha} \cdot \boldsymbol{\alpha} = \\
&= \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{m} y_i y_j \alpha_i \alpha_j \big(\boldsymbol{x_i} \cdot \boldsymbol{x_j} + \frac{1}{C} \delta_{ij} \big),
\end{aligned} \tag{25}$$

where the Kronecker's delta, $\delta_{ij} = 1$ if $i = j$ and 0 otherwise. The objective function above is subject to the usual constraints:

$$\begin{cases} \alpha_i \geq 0, \quad \forall i = 1, .., m \\ \sum_{i=1}^{m} y_i \alpha_i = 0 \end{cases}$$

This dual formulation can be used to find a solution of Problem 22, which extends the applicability of linear functions to classification problems not completely linearly separable. The separability property relates not only to the available class of hypotheses, e.g. linear vs. polynomial functions, but it strictly depends on the adopted features. Their

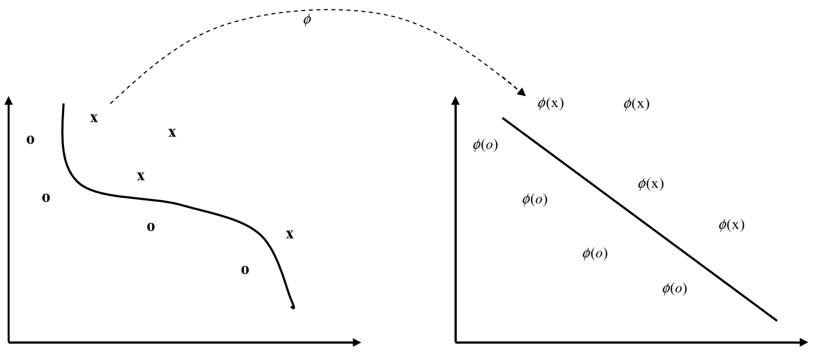


Fig. 16. A mapping ϕ that makes separable the initial data points

roles is to provide a map between the examples and vectors in $\mathbb{R}^n$. Given such mapping, the scalar product provides a measure of the *similarity* between pairs of examples or, according to a more minimalist interpretation, it provides a partitioning function based on such features.

The next section shows that, it is possible to directly substitute the scalar product of two feature vectors with a similarity function between the data examples. This allows for avoiding explicit feature design and consequently enabling the use of similarly measures called kernel functions. These, in turn, define implicit feature spaces.

4 Kernel Methods

One of the most difficult step on applying machine learning is the feature design. Features should represent data in a way that allows learning algorithms to separate positive from negative examples. The features used by SVMs are used to build vector representations of data examples and the scalar product between them. This, sometimes, simply counts the number of common features to measure how much the examples are similar. Instead of encoding data in feature vectors, we may design kernel functions that provide such similarity between examples avoiding the use of explicit feature representations. The reader may object that the learning algorithm still requires the supporting feature space to model the hyperplane and the data points, but this is not necessary if the optimization problem is solved in the dual space.

The real limit of the kernel functions is that they must generate a well defined inner product vector space. Such property will hold if the Mercer's conditions are satisfied. Fortunately, there are many kernels, e.g. polynomial, string, lexical and tree kernels that satisfy such conditions and give us the possibility to use them in SVMs.

Kernels allow for more abstractly defining our leaning problems and in many cases allow for solving non linear problems by re-mapping the initial data points in a separable space as shown by Figure 16. The following example illustrates one of the case in which a non-linear function can be expressed in a linear formulation in a different space.

Example 2. Overcoming linear inseparability
Suppose that we want to study the force of interactions between two masses m_1 and m_2. m_1 is free to move whereas m_2 is blocked. The distance between the two masses is indicated with r and their are subject to the Newtown's gravity law:

$$f(m_1, m_2, r) = C\frac{m_1 m_2}{r^2},$$

Thus mass m_1 naturally tends to move towards m_2.

We apply a force f_a of inverse direction with respect to f to m_1. As a result, we note that sometimes m_1 approaches m_2 whereas other times it gets far from it. To study such phenomenon, we carry out a set of experiments with different experimental parameters, i.e. m_1, m_2, r and f_a and we annotate the result of our action: success if m_1 gets closer to m_2 (or does not move) and failure otherwise.

Each successful experiment can be considered a positive example whereas unsuccessful experiments are considered negative examples. The parameters above constitute feature vectors representing an experiment. We can apply SVMs to learn the classification of new experiments $\langle f_a, m_1, m_2, r\rangle$ in successful or unsuccessful, i.e. if $f_a - f(m_1, m_2, r) \geq 0$ or otherwise, respectively. This means that SVMs have to learn the gravitational law function, $f(m_1, m_2, r)$, but, since this is clearly non-linear, hard margin SVMs will not generally converge and soft margin SVMs will provide inaccurate results.

The solution for this problem is to map our initial feature space in another vector space, i.e. $\langle f_a, m_1, m_2, r\rangle \rightarrow \langle ln f_a, ln(m_1), ln(m_2), ln(r)\rangle = \langle k, x, y, z\rangle$. Since $ln\big(f(m_1, m_2, r)\big) = ln(C) + ln(m_1) + ln(m_2) - 2ln(r) = c + x + y - 2z$, we can express the logarithm of gravity law with a linear combination of the transformed features in the new space. In more detail, points above (or lying on) the ideal hyperplane $k-(c+x+y-2z) = 0$, i.e. points that satisfy $k-(c+x+y-2z) \geq 0$ (or equivalently that satisfy $f_a - f(m_1, m_2, r) \geq 0$), are successful experiments whereas points below such hyperplane are unsuccessful. The above passages prove that a separating hyperplane of the training set always exists in the transformed space, consequently SVMs will always converge (with an error dependent on the number of training examples).

4.1 The Kernel Trick

Section 3.1 has shown that the Perceptron algorithm, used to learn linear classifiers, can be adapted to work in the dual space. In particular, such algorithm (see Table 3) clearly shows that it only exploits feature vectors in the form of scalar product. Consequently, we can replace feature vectors $\boldsymbol{x}_i$ with the data objects o_i, substituting the scalar product $\boldsymbol{x}_i \cdot \boldsymbol{x}_j$ with a kernel function $k(o_i, o_j)$, where o_i are the initial objects mapped into $\boldsymbol{x}_i$ using a feature representation, $\phi(.)$. This implies that $\boldsymbol{x}_i \cdot \boldsymbol{x}_j = \phi(o_i) \cdot \phi(o_j) = k(o_i, o_j)$.

Similarly to the Perceptron algorithm, the dual optimization problem of Soft Margin SVMs (Eq. 25) uses feature vectors only inside a scalar product, which can be substituted with $k(o_i, o_j)$. Therefore, the kernelized version of the soft margin SVMs is

$$\begin{cases} maximize \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{m} y_i y_j \alpha_i \alpha_j \big(k(o_i, o_j) + \frac{1}{C} \delta_{ij}\big) \\ \alpha_i \geq 0, \quad \forall i = 1,..,m \\ \sum_{i=1}^{m} y_i \alpha_i = 0 \end{cases}$$

Moreover, Eq. 10 for the Perceptron appears also in the Soft Margin SVMs (see conditions 24), hence we can rewrite the SVM classification function as in Eq. 11 and use a kernel inside it, i.e.:

$$h(x) = sgn\left(\sum_{i=1}^{m} \alpha_i y_i k(o_i, o_j) + b \right)$$

The data object o is mapped in the vector space trough a feature extraction procedure $\phi : o \rightarrow (x_1, ..., x_n) = \boldsymbol{x}$, more in general, we can map a vector $\boldsymbol{x}$ from one feature space into another one:

$$\boldsymbol{x} = (x_1, ..., x_n) \rightarrow \boldsymbol{\phi}(\boldsymbol{x}) = (\phi_1(\boldsymbol{x}), ..., \phi_n(\boldsymbol{x}))$$

This leads to the general definition of kernel functions:

Definition 10. *A kernel is a function k, such that* $\forall$ $\boldsymbol{x},\boldsymbol{z} \in X$

$$k(\boldsymbol{x}, \boldsymbol{z}) = \boldsymbol{\phi}(\boldsymbol{x}) \cdot \boldsymbol{\phi}(\boldsymbol{z})$$

where $\boldsymbol{\phi}$ *is a mapping from* X *to an (inner product) feature space.*

Note that, once we have defined a kernel function that is effective for a given learning problem, we do not need to find which mapping ϕ corresponds to. It is enough to know that such mapping exists. The following proposition states the conditions that guarantee such existence.

Proposition 1. *(Mercer's conditions)*
Let X be a finite input space and let $K(\boldsymbol{x}, \boldsymbol{z})$ *be a symmetric function on X. Then* $K(\boldsymbol{x}, \boldsymbol{z})$ *is a kernel function if and only if the matrix*

$$k(\boldsymbol{x}, \boldsymbol{z}) = \boldsymbol{\phi}(\boldsymbol{x}) \cdot \boldsymbol{\phi}(\boldsymbol{z})$$

is positive semi-definite (has non-negative eigenvalues).

Proof. Let us consider a symmetric function on a finite space $X = \{x_1, x_2, ..., x_n\}$

$$\boldsymbol{K} = \big(K(x_i, x_j)\big)_{i,j=1}^{n}$$

Since $\boldsymbol{K}$ is symmetric there is an orthogonal matrix $\boldsymbol{V}$ such that $\boldsymbol{K} = \boldsymbol{V \Lambda V'}$ where $\boldsymbol{\Lambda}$ is a diagonal matrix containing the eigenvalues λ_t of $\boldsymbol{K}$, with corresponding eigenvectors $\boldsymbol{v}_t = (v_{ti})_{i=1}^n$, i.e., the columns of $\boldsymbol{V}$. Now assume that all the eigenvalues are non-negatives and consider the feature mapping:

$$\boldsymbol{\phi} : \boldsymbol{x}_i \rightarrow \left(\sqrt{\lambda_t} v_{ti}\right)_{t=1}^n \in \mathbb{R}^n, i = 1, .., n.$$

It follows that

$$\boldsymbol{\phi}(\boldsymbol{x}_i) \cdot \boldsymbol{\phi}(\boldsymbol{x}_j) = \sum_{t=1}^n \lambda_t v_{ti} v_{tj} = (\boldsymbol{V\Lambda V'})_{ij} = \boldsymbol{K}_{ij} = K(x_i, x_j).$$

This proves that $K(\boldsymbol{x}, \boldsymbol{z})$ is a *valid* kernel function that corresponds to the mapping $\boldsymbol{\phi}$. Therefore, the only requirement to derive the mapping $\boldsymbol{\phi}$ is that the eigenvalues of $\boldsymbol{K}$ are non-negatives since if we had a negative eigenvalue λ_s associated with the eigenvector $\boldsymbol{v}_s$, the point

$$\boldsymbol{z} = \sum_{i=1}^n \boldsymbol{v}_{si} \phi(\boldsymbol{x}_i) = \sqrt{\boldsymbol{\Lambda}} \boldsymbol{V}' \boldsymbol{v}_s.$$

in the feature space would have norm squared

$$||\boldsymbol{z}||^2 = \boldsymbol{z} \cdot \boldsymbol{z} = \boldsymbol{v}_s' \boldsymbol{V} \sqrt{\boldsymbol{\Lambda}} \sqrt{\boldsymbol{\Lambda}} \boldsymbol{V}' \boldsymbol{v}_s = \boldsymbol{v}_s' \boldsymbol{V\Lambda V'} \boldsymbol{v}_s = \boldsymbol{v}_s' \boldsymbol{K} \boldsymbol{v}_s = \lambda_s < 0,$$

which contradicts the geometry of the space [20].

4.2 Polynomial Kernel

The above section has shown that kernel functions can be used to map a vector space in other spaces in which the target classification problem becomes linearly separable (or in general easier). Another advantage is the possibility to map the initial feature space in a richer space which includes a high number of dimensions (possibly infinite): this may result in a better description of the objects and higher accuracy. For example, the polynomial kernel maps the initial features in a space which contains both the original features and all the possible feature conjunctions. For example, given the components x_1 and x_2, the new space will contain $x_1 x_2$. This is interesting for text categorization as the polynomial kernel automatically derives the feature `hard rock` or `hard disk` from the individual features `hard`, `rock` and `disk`. The conjunctive features may help to disambiguate between *Music Store* and *Computer Store* categories.

The great advantage of using kernel functions is that we do not need to keep the vectors of the new space in the computer memory to evaluate the inner product. For example, suppose that the initial feature space has a cardinality of 100,000 features, i.e., a typical size of the vocabulary in a text categorization problem, only the number of word pairs would be 10^{10}, which cannot be managed by many learning algorithms. The polynomial kernel can be used to evaluate the scalar product between pairs of vectors of such huge space by only using the initial space and vocabulary, as it is shown by the following passages:

$$\begin{aligned}(\boldsymbol{x}\cdot\boldsymbol{z})^2 &= \Big(\sum_{i=1}^{n} x_i z_i\Big)^2 &&= \Big(\sum_{i=1}^{n} x_i z_i\Big)\Big(\sum_{j=1}^{n} x_i z_i\Big)\\ &= \sum_{i=1}^{n}\sum_{j=1}^{n} x_i x_j z_i z_j &&= \sum_{i,j\in\{1,..,n\}} (x_i x_j)(z_i z_j)\\ &= \sum_{k=1}^{m} X_k Z_k &&= \boldsymbol{X}\cdot\boldsymbol{Z},\end{aligned}$$

where:

- $\boldsymbol{x}$ and $\boldsymbol{z}$ are two vectors of the initial space,
- $\boldsymbol{X}$ and $\boldsymbol{Z}$ are the vectors of the final space and
- $X_k = x_i x_j$, $Z_k = z_i z_j$ with $k = (i-1)\times n + j$ and $m = n^2$.

We note that

- the mapping between the two space is $\phi(\boldsymbol{x}) = (x_i x_j)$ for $j = 1,..,n$ and for $i = 1,..,n$;
- to evaluate $\boldsymbol{X}\cdot\boldsymbol{Z}$, we just compute the square of the scalar product in the initial space, i.e. $(\boldsymbol{x}\cdot\boldsymbol{z})^2$; and
- the final space contains conjunctions and also the features of the initial space ($x_i x_i$ is equivalent to x_i).

Additionally, since $x_i x_j = x_j x_i$, the conjunctions receive the double of the weight of single features. The number of distinct features are: n for $i = 1$ and $j = 1,..,n$; $(n-1)$ for $i = 2$ and $j = 2,..,n$; ..; and 1 for $i = n$ and $j = n$. It follows that the total number of terms is

$$n + (n-1) + (n-2) + .. + 1 = \sum_{k=1}^{n} k = \frac{n(n+1)}{2}$$

Another way to compute such number it to consider that, to build all the monomials, the first variable can be chosen out of $n+1$ possibilities (n symbols to form conjunctions and the empty symbol for the single feature) whereas for the second variable only n chances are available (no empty symbol at this time). This way, we obtain all permutations of each monomial of two variables. To compute the number of distinct features, we can divide the number of monomials, i.e. $(n+1)n$, by the number of permutations of two variables, i.e. $2! = 2$. The final quantity can be expressed with the binomial coefficient $\binom{n+1}{2}$.

Given the above observation, we can generalize the kernel from degree 2 to a degree d by computing $(\boldsymbol{x}\cdot\boldsymbol{z})^d$. The results are all monomials of degree d or equivalently all the conjunctions constituted up to d features. The distinct features will be $\binom{n+d-1}{d}$ since we can choose either the empty symbol up to $d-1$ times or n variables.

A still more general kernel can be derived by introducing a constant in the scalar product computation. Hereafter, we show the case for a degree equal to two:

$$(\boldsymbol{x}\cdot\boldsymbol{z}+c)^2 = \Big(\sum_{i=1}^{n} x_i z_i + c\Big)^2 = \Big(\sum_{i=1}^{n} x_i z_i + c\Big)\Big(\sum_{j=1}^{n} x_i z_i + c\Big) =$$
$$= \sum_{i=1}^{n}\sum_{j=1}^{n} x_i x_j z_i z_j + 2c\sum_{i=1}^{n} x_i z_i + c^2 =$$
$$= \sum_{i,j\in\{1,..,n\}} (x_i x_j)(z_i z_j) + \sum_{i=1}^{n} \big(\sqrt{2c}x_i\big)\big(\sqrt{2c}z_i\big) + c^2$$

Note that the second summation introduces n individual features (i.e. x_i) whose weights are controlled by the parameter c which also determines the strength of the degree 0. Thus, we add (n+1) new features to the $\binom{n+1}{2}$ features of the previous kernel of degree 2. If we consider a generic degree d, i.e. the kernel $(\boldsymbol{x}\cdot\boldsymbol{z}+c)^d$, we will obtain $\binom{n+d-1}{d}$ + $n+d-1$ = $\binom{n+d}{d}$ distinct features (which have at least distinct weights). These are all monomials up to and including the degree d.

4.3 String Kernel

Kernel functions can be also applied to discrete space. As a first example, we show their potentiality on the space of finite strings.

Let Σ be a finite alphabet. A string is a finite sequence of characters from Σ, including the empty sequence. We denote by $|s|$ the length of the string $s = s_1, .., s_{|s|}$, where s_i are symbols, and by st the string obtained by concatenating the strings s and t. The string $s[i:j]$ is the substring $s_i, .., s_j$ of s. We say that u is a subsequence of s, if there exist indices $\boldsymbol{I} = (i_1, ..., i_{|u|})$, with $1 \le i_1 < ... < i_{|u|} \le |s|$, such that $u_j = s_{i_j}$, for $j = 1, ..., |u|$, or $u = s[\boldsymbol{I}]$ for short. The length $l(\boldsymbol{I})$ of the subsequence in s is $i_{|u|} - i_i + 1$. We denote by Σ^* the set of all string

$$\Sigma^* = \bigcup_{n=0}^{\infty} \Sigma^n$$

We now define the feature space, $F = \{u_1, u_2..\} = \Sigma^*$, i.e. the space of all possible substrings. We map a string s in $\mathbb{R}^\infty$ space as follows:

$$\phi_u(s) = \sum_{\boldsymbol{I}:u=s[\boldsymbol{I}]} \lambda^{l(\boldsymbol{I})} \tag{26}$$

for some $\lambda \le 1$. These features measure the number of occurrences of subsequences in the string s, weighting them according to their lengths. Hence, the inner product of the feature vectors for two strings s and t gives a sum over all common subsequences weighted according to their frequency of occurrences and lengths, i.e.

$$K(s,t) = \sum_{u\in\Sigma^*} \phi_u(s)\cdot\phi_u(t) = \sum_{u\in\Sigma^*}\sum_{\boldsymbol{I}:u=s[\boldsymbol{I}]} \lambda^{l(\boldsymbol{I})} \sum_{\boldsymbol{J}:u=t[\boldsymbol{J}]} \lambda^{l(\boldsymbol{J})} =$$

$$= \sum_{u\in\Sigma^*}\sum_{\boldsymbol{I}:u=s[\boldsymbol{I}]}\sum_{\boldsymbol{J}:u=t[\boldsymbol{J}]} \lambda^{l(\boldsymbol{I})+l(\boldsymbol{J})} \tag{27}$$

The above equation defines a class of similarity functions known as string kernels or sequence kernels. These functions are very effective for extracting features from streams. For example, in case of text categorization, they allow the learning algorithm to quantify the matching between two different words, phrases, sentences or whole documents. Given two strings, *Bank* and *Rank*:

– B, a, n, k, Ba, Ban, Bank, an, ank, nk, Bn, Bnk, Bk and ak are substrings of $Bank$.
– R, a, n, k, Ra, Ran, Rank, an, ank, nk, Rn, Rnk, Rk and ak are substrings of $Rank$.

Such substrings are features in the Σ^* that have non-null weights. These are evaluated by means of Eq. 26, e.g. $\phi_{\texttt{B}}(\texttt{Bank}) = \lambda^{(i_1-i_1+1)} = \lambda^{(1-1+1)} = \lambda$, $\phi_{\texttt{k}}(\texttt{Bank}) = \lambda^{(i_1-i_1+1)} = \lambda^{(4-4+1)} = \lambda$, $\phi_{\texttt{an}}(\texttt{Bank}) = \lambda^{(i_2-i_1+1)} = \lambda^{(3-2+1)} = \lambda^2$ and $\phi_{\texttt{Bk}}(\texttt{Bank}) = \lambda^{(i_2-i_1+1)} = \lambda^{(4-1+1)} = \lambda^4$.

Since Eq. 27 requires that the substrings in *Bank* and *Rank* match, we need to evaluate Eq. 26 only for the common substrings, i.e.:

- $\phi_{\texttt{a}}(\texttt{Bank}) = \phi_{\texttt{a}}(\texttt{Rank}) = \lambda^{(i_1-i_1+1)} = \lambda^{(2-2+1)} = \lambda$,
- $\phi_{\texttt{n}}(\texttt{Bank}) = \phi_{\texttt{n}}(\texttt{Rank}) = \lambda^{(i_1-i_1+1)} = \lambda^{(3-3+1)} = \lambda$,
- $\phi_{\texttt{k}}(\texttt{Bank}) = \phi_{\texttt{k}}(\texttt{Rank}) = \lambda^{(i_1-i_1+1)} = \lambda^{(4-4+1)} = \lambda$,
- $\phi_{\texttt{an}}(\texttt{Bank}) = \phi_{\texttt{an}}(\texttt{Rank}) = \lambda^{(i_2-i_1+1)} = \lambda^{(3-2+1)} = \lambda^2$,
- $\phi_{\texttt{ank}}(\texttt{Bank}) = \phi_{\texttt{ank}}(\texttt{Rank}) = \lambda^{(i_3-i_1+1)} = \lambda^{(4-2+1)} = \lambda^3$,
- $\phi_{\texttt{nk}}(\texttt{Bank}) = \phi_{\texttt{nk}}(\texttt{Rank}) = \lambda^{(i_2-i_1+1)} = \lambda^{(4-3+1)} = \lambda^2$,
- $\phi_{\texttt{ak}}(\texttt{Bank}) = \phi_{\texttt{ak}}(\texttt{Rank}) = \lambda^{(i_2-i_1+1)} = \lambda^{(4-2+1)} = \lambda^3$.

It follows that $K(\texttt{Bank}, \texttt{Rank}) = (\lambda, \lambda, \lambda, \lambda^2, \lambda^3, \lambda^2, \lambda^3) \cdot (\lambda, \lambda, \lambda, \lambda^2, \lambda^3, \lambda^2, \lambda^3)$ $= 3\lambda^2 + 2\lambda^4 + 2\lambda^6$.

From this example, we note that short non-discontinuous strings receive the highest contribution, e.g. $\phi_{\texttt{B}}(\texttt{Bank}) = \lambda > \phi_{\texttt{an}}(\texttt{Bank}) = \lambda^2$. This may appear counterintuitive as longer string should be more important to characterize two textual snippets. Such inconsistency disappears if we consider that when a large string is matched, the same will happen for all its substrings. For example, the contribution coming from $Bank$, in the matching between the "$Bank\ of\ America$" and "$Bank\ of\ Italy$" strings, includes the match of B, a, n, k, Ba, Ban,..., an so on.

Moreover, it should be noted that Eq. 27 is rather expensive from a computational viewpoint. A method for its fast computation trough a recursive function was proposed in [38].

First, a kernel over the space of strings of length n, Σ^n is computed, i.e.

$$K_n(s,t) = \sum_{u \in \Sigma^n} \phi_u(s) \cdot \phi_u(t) = \sum_{u \in \Sigma^n} \sum_{\boldsymbol{I}:u=s[\boldsymbol{I}]} \sum_{\boldsymbol{J}:u=t[\boldsymbol{J}]} \lambda^{l(\boldsymbol{I})+l(\boldsymbol{J})}.$$

Second, a slightly different version of the above function is considered, i.e.

$$K_i'(s,t) = \sum_{u \in \Sigma^n} \phi_u(s) \cdot \phi_u(t) = \sum_{u \in \Sigma^i} \sum_{\boldsymbol{I}:u=s[\boldsymbol{I}]} \sum_{\boldsymbol{J}:u=t[\boldsymbol{J}]} \lambda^{|s|+|t|-i_1-j_1+2},$$

for $i = 1,..,n-1$. $K'_i(s,t)$ is different than $K_n(s,t)$ since, to assign weights, the distances from the initial character of the substrings to the end of the string, i.e. $|s| - i_1 + 1$ and $|t| - j_1 + 1$, are used in place of the distances between the first and last characters of the substrings, i.e. $l(\boldsymbol{I})$ and $l(\boldsymbol{J})$.

It can be proved that $K_n(s,t)$ is evaluated by the following recursive relations:

- $K'_0(s,t) = 1$, for all s,t,
- $K'_i(s,t) = 0$, if min $(|s|,|t|) < i$,
- $K_i(s,t) = 0$, if min $(|s|,|t|) < i$,
- $K'_i(sx,t) = \lambda K'_i(s,t) + \sum_{j:t_j=x} K'_{i-1}(s,t[1:j-1])\lambda^{|t|-j+2}, i = 1,..,n-1,$
- $K_n(sx,t) = K_n(s,t) + \sum_{j:t_j=x} K'_{n-1}(s,t[1:j-1])\lambda^2.$

The general idea is that $K'_{i-1}(s,t)$ can be used to compute $K_n(s,t)$ when we increase the size of the input strings of one character, e.g. $K_n(sx,t)$. Indeed, K'_i and K_i compute the same quantity when the last character of the substring $u \in \Sigma^i$, i.e. x, coincides with the last character of the string, i.e. the string can be written as sx. Since $K'_i(sx,t)$ can be reduced to $K'_i(s,t)$, the recursion relation is valid. The computation time of such process is proportional to $n \times |s| \times |t|$, i.e. an efficient evaluation.

4.4 Lexical Kernel

The most used Information Retrieval (IR) paradigm is based on the assumptions that (a) the semantic of a document can be represented by the semantic of its words and (b) to express the similarity between document pairs, it is enough to only consider the contribution from matching terms. In this view, two words that are strongly related, e.g. synonyms, do not contribute with their *relatedness* to the document similarity.

More advanced IR models attempt to take the above problem into account by introducing term similarities. Complex and interesting term similarities can be implemented using external (to the target corpus) thesaurus, like for example the Wordnet hierarchy [26]. For example, the terms *mammal* and *invertebrate* are under the term *animal* in such hierarchy. In turns, the terms *dog* and *cat*, are under the term *mammal*. The length of the path that connects two terms in such hierarchy intuitively provides a sort of similarity metrics. Once a term relatedness is designed, document similarities, which are the core functions of most Text Categorization algorithms, can be designed as well.

Given a term similarity function σ and two documents d_1 and $d_2 \in D$ (the document set), we define their similarity as:

$$K(d_1,d_2) = \sum_{f_1 \in d_1, f_2 \in d_2} (w_1 w_2) \times \sigma(f_1,f_2) \qquad (28)$$

where w_1 and w_2 are the weights of the words (features) f_1 and f_2 in the documents d_1 and d_2, respectively. Interestingly such similarity can be a valid kernel function and, therefore, used in SVMs. To prove this we need to verify the Mercer's conditions, i.e. that the associated kernel matrix (see Proposition 1) is positive semi-definite. We can apply single value decomposition and check the eigenvalues. In case we find that some of them are negative, we can still use the lexical kernel by squaring its associated matrix. Indeed, the kernel $K(d_1, d_2)$ can be written as $\boldsymbol{P} = \boldsymbol{M}' \cdot \boldsymbol{M}$, where $\boldsymbol{M}$ is the matrix defined by $\sigma(f_1, f_2)$ and $\boldsymbol{M}'$ is its transposed. Since $\boldsymbol{P}$ is surely positive semi-definite (it is a square), $K(d_1, d_2) = \boldsymbol{P}$ satisfies the Mercer's conditions.

The lexical kernel has been successfully applied to improve document categorization [8] when few documents are available for training. Indeed, the possibility to match different words using a σ similarity allows SVMs to recover important semantic information.

5 Tree Kernel Spaces

The polynomial and the string kernels have shown that, starting from an initial feature set, they can automatically provide a very high number of interesting features. These are a first example of the usefulness of kernel methods. Other interesting kernel approaches aim to automatically generate large number of features from structures. For example, tree kernels are able to extract many types of tree fragments from a target tree. One of their purposes is to model syntactic information in a target learning problem. In particular, tree kernels seem well suited to model syntax in natural language applications, e.g. for the extraction of semantic predicative structures like *bought(Mary, a cat, in Rome)* [54].

Indeed, previous work shows that defining linguistic theories for the modeling of natural languages (e.g. [35]) is a complex problem, far away from a sound and complete solution, e.g. the links between syntax and semantic are not completely understood yet. This makes the design of syntactic features for the automatic learning of semantic structures complex and consequently both remarkable deep knowledge about the target linguistic phenomena and research effort are required.

Kernel methods, which do not require any noticeable feature design effort, can provide the same accuracy of manually designed features and sometime they can suggest new solutions to the designer to improve the model of the target linguistic phenomenon.

The kernels that we consider in next sections represent trees in terms of their substructures (fragments). Their are based on the general notion of convolution kernels hereafter reported.

Definition 11. *General Convolution Kernels*
Let $X, X_1, .., X_m$ be separable metric spaces, $x \in X$ a structure and $\boldsymbol{x} = x_1, ..., x_m$ its parts, where $x_i \in X_i \quad \forall i = 1, .., m$. Let R be a relation on the set $X \times X_1 \times .. \times X_m$ such that $R(\boldsymbol{x}, x)$ holds if $\boldsymbol{x}$ are the parts of x. We indicate with $R^{-1}(x)$ the set $\{\boldsymbol{x} : R(\boldsymbol{x}, x)\}$. Given two objects x and $y \in X$, their similarity $K(x, y)$ is defined as:

$$K(x,y) = \sum_{\boldsymbol{x} \in R^{-1}(x)} \sum_{\boldsymbol{y} \in R^{-1}(y)} \prod_{i=1}^{m} K_i(x_i, y_i) \tag{29}$$

Subparts or fragments define a feature space which, in turn, is mapped into a vector space, e.g. $\mathbb{R}^n$. In case of tree kernels, the similarity between trees is given by the number of common tree fragments. These functions detect if a common tree subpart

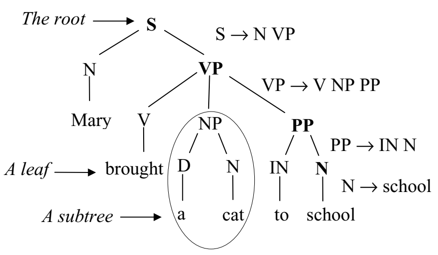


Fig. 17. A syntactic parse tree

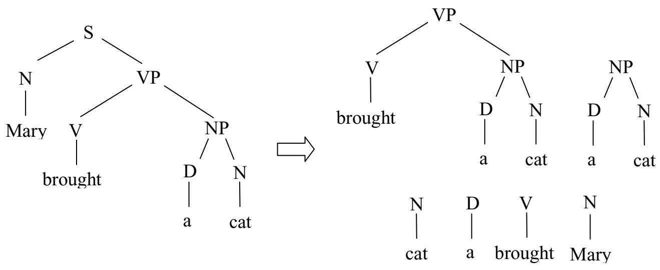


Fig. 18. A syntactic parse tree with its SubTrees (STs)

belongs to the feature space that we intend to generate. For such purpose, the fragment type needs to be described. We consider three important characterizations: the SubTrees (STs), the SubSet Trees (SSTs) and the Partial Trees (PTs).

5.1 SubTree, SubSet Tree and Partial Tree Kernels

Trees are directed, connected acyclic graphs with a special node called root. Their recursive definition is the following: (1) the root node, connected with one or more nodes (called children), is a tree and (2) a child can be a tree, i.e. a SubTree, or a node without children, i.e. a leaf.

In case of syntactic parse trees each node with its children is associated with a grammar production rule, where the symbol at left-hand side corresponds to the parent and the symbols at right-hand side are associated with the children. The terminal symbols of the grammar are always associated with the leaves of the tree. For example, Figure 17 illustrates the syntactic parse of the sentence `"Mary brought a cat to school"`.

We define a **SubTree** (ST) as any node of a tree along with all its descendants. For example, the line in Figure 17 circles the SubTree rooted in the `NP` node. A **SubSet Tree** (SST) is a more general structure which not necessarily includes all its descendants. The only restriction is that an SST must be generated by applying the same grammatical rule set that generated the original tree, as pointed out in [19]. Thus, the difference with the SubTrees is that the SST's leaves can be associated with non-terminal symbols. For example, `[S [N VP]]` is an SST of the tree in Figure 17 and it has the two non-terminal symbols `N` and `VP` as leaves.

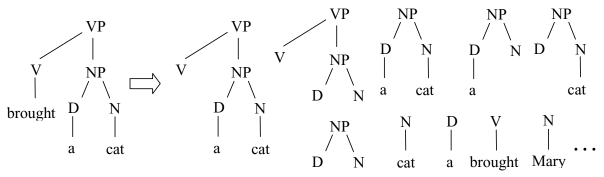


Fig. 19. A tree with some of its SubSet Trees (SSTs)

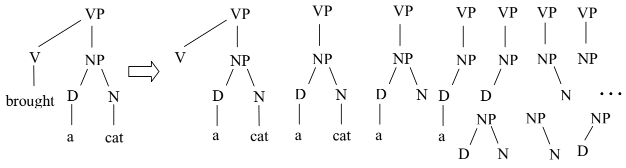


Fig. 20. A tree with some of its Partial Trees (PTs)

If we relax the constraint over the SSTs, we obtain a more general form of substructures that we defined as **Partial Trees** (PTs). These can be generated by the application of partial production rules of the original grammar. For example, `[S [N VP]]`, `[S [N]]` and `[S [VP]]` are valid PTs of the tree in Figure 17.

Given a syntactic tree, we may represent it by means of the set of all its STs, SSTs or PTs. For example, Figure 18 shows the parse tree of the sentence `"Mary brought a cat"` together with its 6 STs. The number of SSTs is always higher. For example, Figure 19 shows 10 SSTs (out of all 17) of the SubTree of Figure 18 rooted in `VP`. Figure 20 shows that the number of PTs derived from the same tree is even higher (i.e. 30 PTs). These different substructure numbers provide an intuitive quantification of the different information level of the diverse tree-based representations.

5.2 The Kernel Functions

The main idea of the tree kernels is to compute the number of the common substructures between two trees T_1 and T_2 without explicitly considering the whole fragment space. For this purpose, we slightly modified the kernel function proposed in [19] by introducing a parameters σ, which enables the ST or the SST evaluation. For the PT kernel function, we designed a new algorithm.

The ST and SST Computation. Given a tree fragment space $\{f_1, f_2, ..\} = \mathcal{F}$, we defined the indicator function $I_i(n)$, which is equal to 1 if the target f_i is rooted at node n and 0 otherwise. It follows that:

$$K(T_1, T_2) = \sum_{n_1 \in N_{T_1}} \sum_{n_2 \in N_{T_2}} \Delta(n_1, n_2) \tag{30}$$

where N_{T_1} and N_{T_2} are the sets of the T_1's and T_2's nodes, respectively and $\Delta(n_1, n_2) = \sum_{i=1}^{|\mathcal{F}|} I_i(n_1) I_i(n_2)$. This latter is equal to the number of common fragments rooted at the n_1 and n_2 nodes. We can compute Δ as follows:

1. if the productions at n_1 and n_2 are different then $\Delta(n_1, n_2) = 0$;
2. if the productions at n_1 and n_2 are the same, and n_1 and n_2 have only leaf children (i.e. they are pre-terminal symbols) then $\Delta(n_1, n_2) = 1$;
3. if the productions at n_1 and n_2 are the same, and n_1 and n_2 are not pre-terminals then

$$\Delta(n_1, n_2) = \prod_{j=1}^{nc(n_1)} (\sigma + \Delta(c^j_{n_1}, c^j_{n_2})) \tag{31}$$

where $\sigma \in \{0, 1\}$, $nc(n_1)$ is the number of the children of n_1 and c^j_n is the j-th child of the node n. Note that, as the productions are the same $nc(n_1) = nc(n_2)$.

When $\sigma = 0$, $\Delta(n_1, n_2)$ is equal 1 only if $\forall j \quad \Delta(c^j_{n_1}, c^j_{n_2}) = 1$, i.e. all the productions associated with the children are identical. By recursively applying this property, it follows that the SubTrees in n_1 and n_2 are identical. Thus, Eq. 30 evaluates the SubTree (ST) kernel. When $\sigma = 1$, $\Delta(n_1, n_2)$ evaluates the number of SSTs common to n_1 and n_2 as proved in [19].

To include the leaves as fragments it is enough to add, to the recursive rule set above, the condition:

0. if n_1 and n_2 are leaves and their associated symbols are equal then $\Delta(n_1, n_2) = 1$

We will refer to such extended kernels as ST+bow (*bag-of-words*) and SST+bow. Moreover, we use the decay factor λ as follows[7]:$\Delta(n_x, n_z) = \lambda$ and $\Delta(n_x, n_z) = \lambda \prod_{j=1}^{nc(n_x)} (\sigma + \Delta(c^j_{n_1}, c^j_{n_2}))$.

The Δ computation complexity is $O(|N_{T_1}| \times |N_{T_2}|)$ time as proved in [19]. We will refer to this basic implementation as the Quadratic Tree Kernel (QTK).

The PT Kernel Function. The evaluation of the Partial Trees is more complex since two nodes n_1 and n_2 with different child sets (i.e. associated with different productions) can share one or more children, consequently they have one or more common substructures, e.g. `[S [DT JJ N]]` and `[S [DT N N]]` have the `[S [N]]` (2 times) and the `[S [DT N]]` in common.

To evaluate all possible substructures common to two trees, we can (1) select a child subset from both trees, (2) extract the portion of the syntactic rule that contains such subset, (3) apply Eq. 31 to the extracted partial productions and (4) sum the contributions of all children subsets.

Such subsets correspond to all possible common (non-continuous) node subsequences and can be computed efficiently by means of sequence kernels [38]. Let $\boldsymbol{J}_1 = (J_{11}, .., J_{1r})$ and $\boldsymbol{J}_2 = (J_{21}, .., J_{2r})$ be the index sequences associate with the ordered child sequences of n_1 and n_2, respectively, then the number of PTs is evaluated by the following Δ function:

$$\Delta(n_1, n_2) = 1 + \sum_{\boldsymbol{J}_1, \boldsymbol{J}_2, l(\boldsymbol{J}_1) = l(\boldsymbol{J}_2)} \prod_{i=1}^{l(\boldsymbol{J}_1)} \Delta(c^{\boldsymbol{J}_{1i}}_{n_1}, c^{\boldsymbol{J}_{2i}}_{n_2}), \tag{32}$$

[7] To have a similarity score between 0 and 1, we also apply the normalization in the kernel space, i.e. $K_{normed}(T_1, T_2) = \frac{K(T_1, T_2)}{\sqrt{K(T_1, T_1) \times K(T_2, T_2)}}$.

where $l(\boldsymbol{J}_1)$ indicates the length of the target child sequence whereas $\boldsymbol{J}_{1i}$ and $\boldsymbol{J}_{2i}$ are the i^{th} children in the two sequences. We note that:

1. Eq. 32 is a convolution kernel [34] (see Definition 11).
2. Given a sequence of common children, $\boldsymbol{J}$, the product in Eq. 32 evaluates the number of common PTs rooted in n_1 and n_2. In these PTs, the children of n_1 and n_2 are all and only those in $\boldsymbol{J}$.
3. By summing the products associated with each sequence we evaluate all possible PTs (the root is included).
4. Tree kernels based on sequences were proposed in [72; 21] but they do not evaluate all tree substructures, i.e. they are not convolution kernels.
5. We can scale down the contribution from the longer sequences by adding two decay factors λ and μ:

$$\Delta(n_1, n_2) = \mu\Big(\lambda + \sum_{\boldsymbol{J}_1,\boldsymbol{J}_2,l(\boldsymbol{J}_1)=l(\boldsymbol{J}_2)} \lambda^{d(\boldsymbol{J}_1)+d(\boldsymbol{J}_2)} \prod_{i=1}^{l(\boldsymbol{J}_1)} \Delta(c_{n_1}^{\boldsymbol{J}_{1i}}, c_{n_2}^{\boldsymbol{J}_{2i}})\Big)$$

   where $d(\boldsymbol{J}_1) = \boldsymbol{J}_{1l(\boldsymbol{J}_1)} - \boldsymbol{J}_{11} + 1$ and $d(\boldsymbol{J}_2) = \boldsymbol{J}_{2l(\boldsymbol{J}_2)} - \boldsymbol{J}_{21} + 1$.

Finally, as the sequence kernels and the Eq. 31 can be efficiently evaluated, the same can be done for Eq. 32. The computational complexity of PTK is $O(p\rho^2|N_{T_1}| \times |N_{T_2}|)$, where p is the largest subsequence of children that we want to consider and ρ is the maximal outdegree observed in the two trees. However, as shown in [40], the average running time tends to be linear for natural language syntactic trees.

6 Conclusions and Advanced Topics

In this chapter we have shown the basic approaches of traditional machine learning such as *Decision Trees* and *Naive Bayes* and we have introduced the basic concepts of the statistical learning theory such as the characterization of learning via the PAC theory and VC-dimension. We have also presented, the Perceptron algorithm to introduce a simplified theory of Support Vector Machines (SVMs) and kernel methods. Regarding the latter, we have shown some of their potentials, e.g. the Polynomial, String, Lexical and Tree kernels by alluding to their application for Natural Language Processing (NLP).

The interested reader, who would like to acquire much more practical knowledge on the use of SVMs and kernel methods can refer to the following publications clustered by topics (mostly from NLP): *Text Categorization* [9; 56; 10; 6; 5; 11; 12; 7; 13]; *Corefernce Resolution* [66; 65]; *Question Answering* [51; 13; 14; 55; 49; 50]; *Shallow Semantic Parsing* [54; 32; 45; 3; 30; 46; 31; 48; 42; 47; 57; 22; 44]; *Concept segmentation and labeling of text and speech* [23; 24; 59; 36; 37; 33]; *Relational Learning* [68; 69; 52; 67; 70; 71; 58; 43; 39; 27]; *SVM optimization* [40; 1; 41; 2; 53; 60; 61; 63; 62]; *Mapping Natural Language to SQL* [28; 29]; *Protein Classification* [17; 18]; *Audio classification* [4]; and *Electronic Device Failure detection* [25].

The articles above are available at `http://disi.unitn.it/moschitti/Publications.htm` whereas complementary training material can be found at `http://disi.unitn.it/moschitti/teaching.html`. Additionally, SVM software comprising several structural kernels can be downloaded from `http://disi.unitn.it/moschitti/Tree-Kernel.htm`.

Acknowledgement

The work for this tutorial has been partially founded by the European Coordinate Action, EternalS, Trustworthy Eternal Systems via Evolving Software, Data and Knowledge (project number FP7 247758).

I would like to thank Roberto Basili for his contribution to an early draft of this chapter.

References

1. Aiolli, F., Martino, G.D.S., Moschitti, A., Sperduti, A.: Fast On-line Kernel Learning for Trees. In: Proceedings Sixth International Conference on Data Mining, ICDM 2006. IEEE, Los Alamitos (2006)
2. Aiolli, F., Martino, G.D.S., Moschitti, A., Sperduti, A.: Efficient Kernel-based Learning for Trees. In: IEEE Symposium on Computational Intelligence and Data Mining, pp. 308–316. IEEE, Stati Uniti d'America (2007)
3. Ana-Maria, G., Moschitti, A.: Towards Free-text Semantic Parsing: A Unified Framework Based on FrameNet, VerbNet and PropBank. In: The Workshop on Learning Structured Information for Natural Language Applications. EACL (2006)
4. Annesi, P., Basili, R., Gitto, R., Moschitti, A., Petitti, R.: Audio Feature Engineering for Automatic Music Genre Classification. In: RIAO, Paris, France, pp. 702–711 (2007)
5. Basili, R., Cammisa, M., Moschitti, A.: A Semantic Kernel to Exploit Linguistic Knowledge. In: Bandini, S., Manzoni, S. (eds.) AI*IA 2005. LNCS (LNAI), vol. 3673, pp. 290–302. Springer, Heidelberg (2005)
6. Basili, R., Cammisa, M., Moschitti, A.: Effective use of WordNet Semantics via Kernel-based Learning. In: Proceedings of the Ninth Conference on Computational Natural Language Learning, pp. 1–8. The Association for Computational Linguistics (June 2005)
7. Basili, R., Cammisa, M., Moschitti, A.: A semantic Kernel to Classify Texts with very few Training Examples. Informatica, an International Journal of Computing and Informatics 1, 1–10 (2006)
8. Basili, R., Cammisa, M., Moschitti, A.: Effective use of wordnet semantics via kernel-based learning. In: Proceedings of Ninth Conference on Computational Natural Language Learning, Ann Arbor, Michigan USA, June 29-30 (2005)
9. Basili, R., Moschitti, A.: NLP-driven IR: Evaluating Performance over a Text Classification Task. In: International Joint Conference of Artificial Intelligence (2001)
10. Basili, R., Moschitti, A.: Automatic Text Categorization: from Information Retrieval to Support Vector Learning. Aracne Publisher (2005)
11. Basili, R., Moschitti, A., Pazienza, M.T.: Extensive Evaluation of Efficient NLP-driven Text Classification. Applied Artificial Intelligence (2006)
12. Bloehdorn, S., Basili, R., Cammisa, M., Moschitti, A.: Semantic Kernels for Text Classification based on Topological Measures of Feature Similarity. In: Sixth International Conference on Data Mining, ICDM 2006, pp. 808–812. IEEE, Los Alamitos (2006)

13. Bloehdorn, S., Moschitti, A.: Combined Syntactic and Semanitc Kernels for Text Classification. In: Amati, G., Carpineto, C., Romano, G. (eds.) ECiR 2007. LNCS, vol. 4425, pp. 307–318. Springer, Heidelberg (2007)
14. Bloehdorn, S., Moschitti, A.: Exploiting Structure and Semantics for Expressive Text Kernels. In: Proceedings of the Sixteenth ACM Conference on Information and Knowledge Management, CIKM 2007, pp. 861–864. ACM, New York (2007)
15. Blumer, A., Ehrenfeucht, A., Haussler, D., Warmuth, M.K.: Learnability and the vapnik-chervonenkis dimension. Journal of the Association for Computing Machinery 36(4), 929–965 (1989)
16. Burges, C.J.C.: A tutorial on support vector machines for pattern recognition. Data Mining and Knowledge Discovery 2(2), 121–167 (1998)
17. Cilia, E., Moschitti, A.: Advanced Tree-based Kernels for Protein Classification. In: Basili, R., Pazienza, M.T. (eds.) AI*IA 2007. LNCS (LNAI), vol. 4733, pp. 218–229. Springer, Heidelberg (2007)
18. Cilia, E., Moschitti, A., Ammendola, S., Basili, R.: Structured kernels for automatic detection of protein active sites. In: Mining and Learning with Graphs Workshop (MLG) (2006)
19. Collins, M., Duffy, N.: New ranking algorithms for parsing and tagging: Kernels over discrete structures, and the voted perceptron. In: ACL 2002 (2002)
20. Cristianini, N., Shawe-Taylor, J.: An introduction to Support Vector Machines. Cambridge University Press, Cambridge (2000)
21. Culotta, A., Sorensen, J.: Dependency Tree Kernels for Relation Extraction. In: Proceedings of the 42nd Meeting of the Association for Computational Linguistics (ACL 2004), Main Volume, Barcelona, Spain, pp. 423–429 (July 2004)
22. Diab, M., Moschitti, A., Pighin, D.: Semantic Role Labeling Systems for Arabic Language using Kernel Methods. In: Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies, pp. 798–806. Association for Computational Linguistics, Columbus (June 2008)
23. Dinarelli, M., Moschitti, A., Riccardi, G.: Re-Ranking Models for Spoken Language Understanding. In: Proceedings of the 12th Conference of the European Chapter of the ACL (EACL 2009), pp. 202–210. Association for Computational Linguistics, Athens (March 2009)
24. Dinarelli, M., Moschitti, A., Riccardi, G.: Re-Ranking Models Based-on Small Training Data for Spoken Language Understanding. In: Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing, pp. 1076–1085. Association for Computational Linguistics (2009)
25. Dutta, H., Waltz, D., Moschitti, A., Pighin, D., Gross, P., Monteleoni, C., Salleb-Aouissi, A., Boulanger, A., Pooleery, M., Anderson, R.: Estimating the Time Between Failures of Electrical Feeders in the New York Power Grid. In: Next Generation Data Mining Summit, NGDM 2009, Baltimore, MD (2009)
26. Fellbaum, C.: WordNet: An Electronic Lexical Database. MIT Press, Cambridge (1998)
27. Giannone, C., Basili, R., Naggar, P., Moschitti, A.: Supervised Semantic Relation Mining from Linguistically Noisy Text Documents. International Journal on Document Analysis and Recognition 2010, 1–25 (2010)
28. Giordani, A., Moschitti, A.: Semantic Mapping Between Natural Language Questions and SQL Queries via Syntactic Pairing. In: Horacek, H., Métais, E., Muñoz, R., Wolska, M. (eds.) NLDB 2009. LNCS, vol. 5723, pp. 207–221. Springer, Heidelberg (2010)
29. Giordani, A., Moschitti, A.: Syntactic Structural Kernels for Natural Language Interfaces to Databases. In: Buntine, W., Grobelnik, M., Mladenić, D., Shawe-Taylor, J. (eds.) ECML PKDD 2009. LNCS, vol. 5781, pp. 391–406. Springer, Heidelberg (2009)

30. Giuglea, A., Moschitti, A.: Semantic Role Labeling via FrameNet, VerbNet and PropBank. In: COLING-ACL 2006: 21st International Conference on Computational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics, pp. 929–936. Association for Computational Linguistics (July 2006)
31. Giuglea, A.M., Moschitti, A.: Shallow Semantic Parsing Based on FrameNet, VerbNet and PropBank. In: ECAI 2006, 17th Conference on Artificial Intelligence, including Prestigious Applications of Intelligent Systems (PAIS 2006), Riva del Garda, Italy, August 29-September 1. IOS, Amsterdam (2006)
32. Giuglea, A.M., Moschitti, A.: Knowledge Discovery using FrameNet, VerbNet and PropBank. In: Meyers, A. (ed.) Workshop on Ontology and Knowledge Discovering at ECML 2004, Pisa, Italy (2004)
33. Hahn, S., Dinarelli, M., Raymond, C., Lefevre, F., Lehnen, P., Mori, R.D., Moschitti, A., Ney, H., Riccardi, G.: Comparing Stochastic Approaches to Spoken Language Understanding in Multiple Languages. IEEE Transaction on Audio, Speech and Language Processing PP (99), 1–15 (2010)
34. Haussler, D.: Convolution Kernels on Discrete Structures. Technical report ucs-crl-99-10, University of California Santa Cruz (1999)
35. Jackendoff, R.: Semantic Structures. Current Studies in Linguistics series. The MIT Press, Cambridge (1990)
36. Johansson, R., Moschitti, A.: Reranking Models in Fine-grained Opinion Analysis. In: Proceedings of the 23rd International Conference on Computational Linguistics (Coling 2010), Beijing, China, pp. 519–527 (August 2010)
37. Johansson, R., Moschitti, A.: Syntactic and Semantic Structure for Opinion Expression Detection. In: Proceedings of the Fourteenth Conference on Computational Natural Language Learning, Sweden, pp. 67–76 (July 2010)
38. Lodhi, H., Saunders, C., Shawe-Taylor, J., Cristianini, N., Watkins, C.: Text classification using string kernels. In: NIPS, pp. 563–569 (2000)
39. Mehdad, Y., Moschitti, A., Zanzotto, F.: Syntactic/Semantic Structures for Textual Entailment Recognition. In: Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics, pp. 1020–1028. Association for Computational Linguistics, Los Angeles (June 2010)
40. Moschitti, A.: Efficient Convolution Kernels for Dependency and Constituent Syntactic Trees. In: Fürnkranz, J., Scheffer, T., Spiliopoulou, M. (eds.) ECML 2006. LNCS (LNAI), vol. 4212, pp. 318–329. Springer, Heidelberg (2006)
41. Moschitti, A.: Making tree kernels practical for natural language learning. In: EACL 2006: 11th Conference of the European Chapter of the Association for Computational Linguistics. ACL (2006)
42. Moschitti, A.: Syntactic Kernels for Natural Language Learning: the Semantic Role Labeling Case. In: Human Language Technology Conference of the North American Chapter of the Association of Computational Linguistics, pp. 97–100. ACL (2006)
43. Moschitti, A.: Syntactic and Semantic Kernels for Short Text Pair Categorization. In: Proceedings of the 12th Conference of the European Chapter of the ACL (EACL 2009), pp. 576–584. Association for Computational Linguistics, Athens (March 2009)
44. Moschitti, A.: LivingKnowledge: Kernel Methods for Relational Learning and Semantic Modeling. In: Margaria, T., Steffen, B. (eds.) ISoLA 2010, Part II. LNCS, vol. 6416, pp. 15–19. Springer, Heidelberg (2010)
45. Moschitti, A., Giuglea, A.M., Coppola, B., Basili, R.: Hierarchical Semantic Role Labeling. In: Proceedings of the Ninth Conference on Computational Natural Language Learning (CoNLL 2005), June 30, pp. 201–204. Association for Computational Linguistics (2005)

46. Moschitti, A., Pighin, D., Basili, R.: Semantic Role Labeling via Tree Kernel Joint Inference. In: Proceedings of the 10th Conference on Computational Natural Language Learning, pp. 61–68. Association for Computational Linguistics (June 2006)
47. Moschitti, A., Pighin, D., Basili, R.: Tree Kernel Engineering for Proposition Reranking. In: MLG 2006: Proceedings of the International Workshop on Mining and Learning with Graphs (in conjunction with ECML/PKDD 2006), pp. 165–172 (September 2006)
48. Moschitti, A., Pighin, D., Basili, R.: Tree Kernel Engineering in Semantic Role Labeling Systems. In: EACL 2006: 11th Conference of the European Chapter of the Association for Computational Linguistics: Proceedings of the Workshop on Learning Structured Information in Natural Language Applications, pp. 49–56 (2006)
49. Moschitti, A., Quarteroni, S.: Kernels on Linguistic Structures for Answer Extraction. In: 46th Conference of the Association for Computational Linguistics, pp. 113–116. ACL, Columbus (2008)
50. Moschitti, A., Quarteroni, S.: Linguistic Kernels for Answer Re-ranking in Question Answering Systems. Information Processing & Management 2010, 1–36 (2010)
51. Moschitti, A., Quarteroni, S., Basili, R., Manandhar, S.: Exploiting Syntactic and Shallow Semantic Kernels for Question/Answer Classification. In: Proceedings of the 45th Annual Meeting of the Association for Computational Linguistics, pp. 776–783. Association for Computational Linguistics, USA (2007)
52. Moschitti, A., Zanzotto, F.M.: Experimenting a General Purpose Textual Entailment Learner in AVE. In: Peters, C., Clough, P., Gey, F.C., Karlgren, J., Magnini, B., Oard, D.W., de Rijke, M., Stempfhuber, M. (eds.) CLEF 2006. LNCS, vol. 4730, pp. 510–517. Springer, Heidelberg (2007)
53. Moschitti, A., Zanzotto, F.M.: Fast and effective kernels for relational learning from texts. In: Proceedings of the 24th Annual International Conference on Machine Learning, pp. 649–656. ACM, New York (June 2007)
54. Moschitti, A.: A study on convolution kernel for shallow semantic parsing. In: Proceedings of the 42th Conference on Association for Computational Linguistic (ACL 2004), Barcelona, Spain (2004)
55. Moschitti, A.: Kernel Methods, Syntax and Semantics for Relational Text Categorization. In: Proceeding of ACM 17th Conf. on Information and Knowledge Management (CIKM 2008), Napa Valley, CA, USA (2008)
56. Moschitti, A., Basili, R.: Complex Linguistic Features for Text Classification: a Comprehensive Study. In: McDonald, S., Tait, J.I. (eds.) ECIR 2004. LNCS, vol. 2997, pp. 181–196. Springer, Heidelberg (2004)
57. Moschitti, A., Pighin, D., Basili, R.: Tree Kernels for Semantic Role Labeling. Computational Linguistics, 193–224 (2008)
58. Nguyen, T., Moschitti, A., Riccardi, G.: Convolution Kernels on Constituent, Dependency and Sequential Structures for Relation Extraction. In: Proceedings of the Conference on Empirical Methods in Natural Language Processing, pp. 1378–1387. Association for Computational Linguistics, Singapore (August 2009)
59. Nguyen, T.V.T., Moschitti, A., Riccardi, G.: Kernel-based Reranking for Named-Entity Extraction. In: Coling 2010: Posters, Beijing, China, pp. 901–909 (August 2010)
60. Pighin, D., Moschitti, A.: Efficient Linearization of Tree Kernel Functions. In: Proceedings of the Thirteenth Conference on Computational Natural Language Learning (CoNLL 2009), pp. 30–38. Association for Computational Linguistics (2009)
61. Pighin, D., Moschitti, A.: Reverse Engineering of Tree Kernel Feature Spaces. In: Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing, pp. 111–120. Association for Computational Linguistics (2009)

62. Pighin, D., Moschitti, A.: On Reverse Feature Engineering of Syntactic Tree Kernels. In: Proceedings of the Fourteenth Conference on Computational Natural Language Learning, pp. 223–233. Association for Computational Linguistics, Uppsala (July 2010)
63. Severyn, A., Moschitti, A.: Large-Scale Support Vector Learning with Structural Kernels. In: Balcázar, J.L., Bonchi, F., Gionis, A., Sebag, M. (eds.) ECML PKDD 2010, Part III. LNCS, vol. 6323, pp. 229–244. Springer, Heidelberg (2010)
64. Vapnik, V.: The Nature of Statistical Learning Theory. Springer, Heidelberg (1995)
65. Versley, Y., Ponzetto, S.P., Poesio, M., Eidelman, V., Jern, A., Smith, J., Yang, X., Moschitti, A.: BART: A Modular Toolkit for Coreference Resolution. In: ACL (Demo Papers), pp. 9–12 (2008)
66. Vesley, Y., Moschitti, A., Poesio, M.: Coreference Systems based on Kernels Methods. In: International Conference on Computational Linguistics, pp. 961–968. Association for Computational Linguistics (2008)
67. Zanzotto, F.M., Moschitti, A.: Automatic Learning of Textual Entailments with Cross-Pair Similarities. In: The Joint 21st International Conference on Computational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics (COLING-ACL). Association for Computational Linguistics, Sydney (2006)
68. Zanzotto, F.M., Moschitti, A.: Similarity between Pairs of Co-indexed Trees for Textual Entailment Recognition. In: The TextGraphs Workshop at Human Language Technology. Association for Computational Linguistics (2006)
69. Zanzotto, F.M., Moschitti, A., Pennacchiotti, M., Pazienza, M.T.: Learning Textual Entailment from Examples. In: The Second Recognising Textual Entailment Challenge. The Second Recognising Textual Entailment Challenge (2006)
70. Zanzotto, F.M., Pennacchiotti, M., Moschitti, A.: Shallow Semantics in Fast Textual Entailment Rule Learners. In: The Third Recognising Textual Entailment Challenge, pp. 72–77. Association for Computational Linguistics (2007)
71. Zanzotto, F.M., Pennacchiotti, M., Moschitti, A.: A Machine Learning Approach to Recognizing Textual Entailment. Natural Language Engineering 15(4), 551–582 (2009)
72. Zelenko, D., Aone, C., Richardella, A.: Kernel methods for relation extraction. Journal of Machine Learning Research (2003)

Modelling Secure Systems Evolution: Abstract and Concrete Change Specifications

Jan Jürjens[1,2] , Martín Ochoa[1], Holger Schmidt[1], Loïc Marchal[3], Siv Hilde Houmb[4], and Shareeful Islam[5]

[1] Software Engineering, Dep. of Computer Science, TU Dortmund, Germany
[2] Fraunhofer ISST, Germany
[3] Hermès Engineering, Belgium
[4] Secure-NOK AS, Norway
[5] School of Computing, IT and Engineering, University of East London, UK
{jan.jurjens,martin.ochoa,holger.schmidt}@cs.tu-dortmund.de,
loic.marchal@hermes-ecs.com,
sivhoumb@securenok.com,
shareeful@uel.ac.uk

Abstract. Developing security-critical systems is difficult, and there are many well-known examples of vulnerabilities exploited in practice. In fact, there has recently been a lot of work on methods, techniques, and tools to improve this situation already at the system specification and design. However, security-critical systems are increasingly long-living and undergo evolution throughout their lifetime. Therefore, a secure software development approach that supports maintaining the needed levels of security even through later software evolution is highly desirable. In this chapter, we recall the UMLsec approach to model-based security and discuss on tools and techniques to model and verify evolution of UMLsec models.

Keywords: Software Evolution, UMLsec, UMLseCh, Security.

1 Introduction

Typically, a systematic approach focused on software quality – the degree to which a software system meets its requirements – is addressed during design time through design processes and supporting tools. Once the system is put in operation, maintenance and re-engineering operations are supposed to keep it running.

At the same time, successful software-based systems are becoming increasingly long-living [21]. This was demonstrated strikingly with the occurrence of the year 2000 bug, which occurred because software had been in use for far longer than its expected lifespan. Also, software-based systems are getting increasingly security-critical since software now pervades the whole critical infrastructures dealing with critical data of both nations and also private individuals. There is therefore a growing demand for more assurance and verifiable secure IT systems

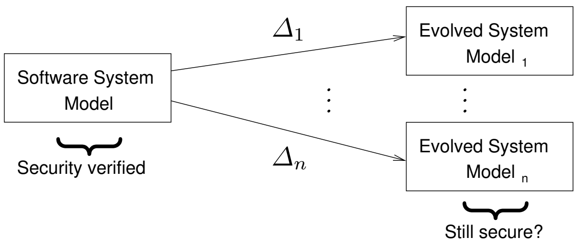


Fig. 1. Model verification problem for n possible evolution paths

both during development and at deployment time, in particular also for long living systems. Yet a long lived system also needs to be flexible, to adapt to evolving requirements, usage, and attack models. However, using today's system engineering techniques we are forced to trade flexibility for assurance or vice versa: we know today how to provide security or flexibility taken in isolation. We can use full fledged verification for providing a high-level of assurance to fairly static requirements, or we can provide flexible software, developed in weeks using agile methodologies, but without any assurance. This raises the research challenge of whether and how we can provide some level of security assurance for something that is going to change.

Our objective is thus to develop techniques and tools that ensure "lifelong" compliance to evolving security requirements for a long-running evolving software system. This is challenging because these requirements are not necessarily preserved by system evolution [22]. In this chapter, we present results towards a security modelling notation for the evolution of security-critical designs, suitable by verification with formally founded automated security analysis tools. Most existing assessment methods used in the development of secure systems are mainly targeted at analysing a static picture of the system or infrastructure in question. For example, the system as it is at the moment, or the system as it will be when we have updated certain parts according to some specifications. In the development of secure systems for longevity, we also need descriptions and specifications of what may be foreseen as future changes, and the assessment methods must be specialized account for this kind of descriptions. Consequently, our approach allows to re-assess the impact that changes might have on the security of systems.

On one hand, a system needs to cope with a given change as early as possible and on the other hand, it should preserve the security properties of the overall system (Fig. 1). To achieve this, it is preferable to analyse the planned evolution before carrying it out. In this chapter we present a notation that allows to precisely determine the changes between one or more system versions, and that combined with proper analysis techniques, allows to reason about how to preserve the existing and new (if any) security properties due to the evolution. Reflecting change on the model level eases system evolution by ensuring effective control and tracking

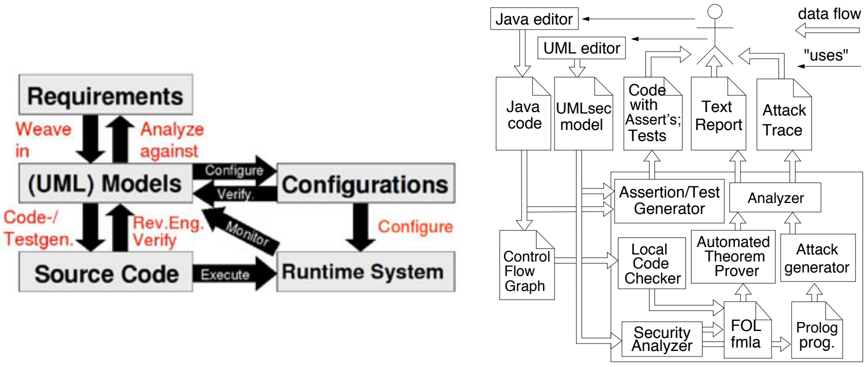


Fig. 2. a) Model-based Security Engineering; b) Model-based Security Tool Suite

of changes. We focus in understanding and tracing the change notations for the system design model. Design models represent the early exploration of the solution space and are the intermediate between requirements and implementation. They can be used to specify, analyse, and trace changes directly.

In Sect. 2 we recall the *UMLsec* profile [10,13], which is a UML [28] lightweight extension to develop and analyse security models, together with some applications. We present a change-modelling profile called *UMLseCh* in Sect. 3. We use UMLseCh design models for change exploration and decision support when considering how to integrate new or additional security functions and to explore the security implications of planned system evolution. To maintain the security properties of a system through change, the change can be explicitly expressed such that its implications can be analysed a priori. The UMLseCh stereotypes extend the existing UMLsec stereotypes so that the design models preserve the security properties due to change.

Although, the question of model evolution is intrinsically related with model-transformation, we do not aim to show an alternative for any existing general-purpose evolution specification or model transformation approaches (such as [7,1,2,25,20]). However, we rely on UMLsec because it comes with sophisticated tool support[1], and our goal is to present an approach that is a) consistent with the UMLsec philosophy of extending UML b) is meant to be used on the UML fragment relevant for UMLsec.

In Sect. 4 we show some applications of UMLseCh to different diagram types and we discuss how this notation and related verification mechanisms could be supported by an extension of the UMLsec Tool Suite.

2 Background: Secure Systems Modelling with UMLsec

Generally, when using model-based development (Fig. 2a), the idea is that one first constructs a model of the system. Then, the implementation is derived from

[1] UMLsec tool suite: `http://www.umlsec.de/`

the model: either automatically using code generation, or manually, in which case one can generate test sequences from the model to establish conformance of the code regarding the model. In the model-based security engineering (MBSE) approach based on the UML [28] extension UMLsec, [11,13], recurring security requirements (such as secrecy, integrity, authenticity, and others) and security assumptions on the system environment, can be specified either within UML specifications, or within the source code (Java or C) as annotations (Fig. 2b). This way we encapsulate knowledge on prudent security engineering as annotations in models or code and make it available to developers who may not be security experts.

The UMLsec extension is given in form of a UML profile using the standard UML extension mechanisms. *Stereotypes* are used together with *tags* to formulate the security requirements and assumptions. *Constraints* give criteria that determine whether the requirements are met by the system design, by referring to a precise semantics of the used fragment of UML. The security-relevant information added using stereotypes includes security assumptions on the physical level of the system, security requirements related to the secure handling and communication of data, and security policies that system parts are supposed to obey. The UMLsec tool-support in Fig. 2b) can be used to check the constraints associated with UMLsec stereotypes mechanically, based on XMI output of the diagrams from the UML drawing tool in use [29,14,18,8]. There is also a framework for implementing verification routines for the constraints associated with the UMLsec stereotypes. Thus advanced users of the UMLsec approach can use this framework to implement verification routines for the constraints of self-defined stereotypes. The semantics for the fragment of UML used for UMLsec is defined in [13] using so-called *UML Machines*, which is a kind of state machine with input/output interfaces similar to Broy's Focus model, whose behavior can be specified in a notation similar to that of Abstract State Machines (ASMs), and which is equipped with UML-type communication mechanisms. On this basis, important security requirements such as secrecy, integrity, authenticity, and secure information flow are defined. To support stepwise development, we show secrecy, integrity, authenticity, and secure information flow to be *preserved* under refinement and the composition of system components. We have also developed an approach that supports the secure development of layered security services (such as layered security protocols). UMLsec can be used to specify and implement security patterns, and is supported by dedicated secure systems development processes, in particular an Aspect-Oriented Modeling approach which separates complex security mechanisms (which implement the security aspect model) from the core functionality of the system (the primary model) in order to allow a security verification of the particularly security-critical parts, and also of the composed model.

2.1 The UMLsec Profile

Because of space restrictions, we cannot recall our formal semantics here completely. Instead, we define precisely and explain the interfaces of this semantics that we need here to define the UMLsec profile. More details on the formal semantics of a simplified fragment of UML and on previous and related work in this area can be found in [9,13]. The semantics is defined formally using so-called *UML Machines*, which is an extension of Mealy automata with UML-type communication mechanisms. It includes the following kinds of diagrams:

Class diagrams define the static class structure of the system: classes with attributes, operations, and signals and relationships between classes. On the instance level, the corresponding diagrams are called *object diagrams*.

Statechart diagrams (or *state diagrams*) give the dynamic behavior of an individual object or component: events may cause a change in state or an execution of actions.

Sequence diagrams describe interaction between objects or system components via message exchange.

Activity diagrams specify the control flow between several components within the system, usually at a higher degree of abstraction than statecharts and sequence diagrams. They can be used to put objects or components in the context of overall system behavior or to explain use cases in more detail.

Deployment diagrams describe the physical layer on which the system is to be implemented.

Subsystems (a certain kind of *packages*) integrate the information between the different kinds of diagrams and between different parts of the system specification.

There is another kind of diagrams, the use case diagrams, which describe typical interactions between a user and a computer system. They are often used in an informal way for negotiation with a customer before a system is designed. We will not use it in the following. Additionally to sequence diagrams, there are *collaboration diagrams*, which present similar information. Also, there are *component diagrams*, presenting part of the information contained in deployment diagrams.

The used fragment of UML is simplified to keep automated formal verification that is necessary for some of the more subtle security requirements feasible. Note that in our approach we identify system objects with UML objects, which is suitable for our purposes. Also, as with practically all analysis methods, also in the real-time setting [30], we are mainly concerned with instance-based models. Although, simplified, our choice of a subset of UML is reasonable for our needs, as we have demonstrated in several industrial case-studies (some of which are documented in [13]).

The formal semantics for subsystems incorporates the formal semantics of the diagrams contained in a subsystem. It

- models actions and internal activities explicitly (rather than treating them as atomic given events), in particular the operations and the parameters employed in them,
- provides passing of messages with their parameters between objects or components specified in different diagrams, including a dispatching mechanism for events and the handling of actions, and thus

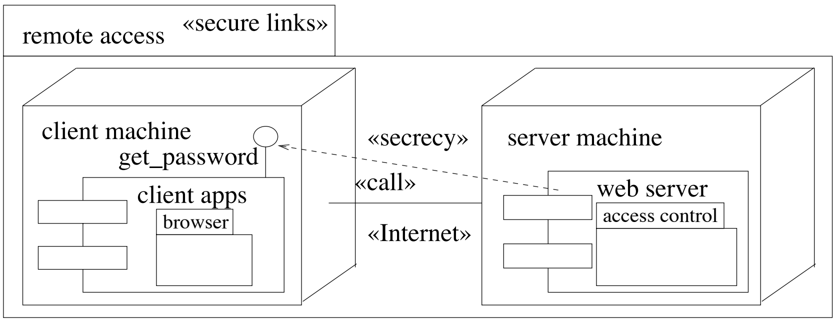


Fig. 3. Example *secure links* usage

– allows in principle whole specification documents to be based on a formal foundation.

In particular, we can compose subsystems by including them into other subsystems.

For example, consider the following UMLsec Stereotype:

secure links. This stereotype, which may label (instances of) subsystems, is used to ensure that security requirements on the communication are met by the physical layer. More precisely, the constraint enforces that for each dependency d with stereotype $s \in \{\ll\texttt{secrecy}\gg, \ll\texttt{integrity}\gg, \ll\texttt{high}\gg\}$ between subsystems or objects on different nodes n, m, we have a communication link l between n and m with stereotype t such that

– in the case of $s = \ll\texttt{high}\gg$, we have $\mathsf{Threats}_A(t) = \emptyset$,
– in the case of $s = \ll\texttt{secrecy}\gg$, we have $\mathsf{read} \notin \mathsf{Threats}_A(t)$, and
– in the case of $s = \ll\texttt{integrity}\gg$, we have $\mathsf{insert} \notin \mathsf{Threats}_A(t)$.

Example. In Fig. 3, given the *default* adversary type, the constraint for the stereotype $\ll\texttt{secure links}\gg$ is violated: The model does not provide communication secrecy against the *default* adversary, because the Internet communication link between web-server and client does not give the needed security level according to the $\mathsf{Threats}_{default}(Internet)$ scenario. Intuitively, the reason is that Internet connections do not provide secrecy against default adversaries. Technically, the constraint is violated, because the dependency carries the stereotype $\ll\texttt{secrecy}\gg$, but for the stereotype $\ll\texttt{Internet}\gg$ of corresponding link we have $\mathsf{read} \in \mathsf{Threats}_{default}(Internet)$.

Code Security Assurance [15,16]. Even if specifications exist for the implemented system, and even if these are formally analyzed, there is usually no guarantee that the implementation actually conforms to the specification. To deal with this problem, we use the following approach: After specifying the system in UMLsec and verifying the model against the given security goals as explained above, we make sure that the implementation correctly implements the specification with techniques explained below. In particular, this approach is applicable to legacy systems. In ongoing work, we are automating this approach to free one of the need to manually construct the UMLsec model.

Run-time Security Monitoring using Assertions. A simple and effective alternative is to insert security checks generated from the UMLsec specification that remain in the code while in use, for example using the assertion statement that is part of the Java language. These assertions then throw security exceptions when violated at run-time. In a similar way, this can also be done for C code.

Model-based Test Generation. For performance-intensive applications, it may be preferable not to leave the assertions active in the code. This can be done by making sure by extensive testing that the assertions are always satisfied. We can generate the test sequences automatically from the UMLsec specifications. More generally, this way we can ensure that the code actually conforms to the UMLsec specification. Since complete test coverage is often infeasible, our approach automatically selects those test cases that are particularly sensitive to the specified security requirements [19].

Automated Code Verification against Interface Specifications. For highly non-deterministic systems such as those using cryptography, testing can only provide assurance up to a certain degree. For higher levels of trustworthiness, it may therefore be desirable to establish that the code does enforce the annotations by a formal verification of the source code against the UMLsec interface specifications. We have developed an approach that does this automatically and efficiently by proving locally that the security checks in the specification are actually enforced in the source code.

Automated Code Security Analysis. We developed an approach to use automated theorem provers for first-order logic to directly formally verify crypto-based Java implementations based on control flow graphs that are automatically generated (and without first manually constructing an interface specification). It supports an abstract and modular security analysis by using assertions in the source code. Thus large software systems can be divided into small parts for which a formal security analysis can be performed more easily and the results composed. Currently, this approach works especially well with nicely structured code (such as created using the MBSE development process).

Secure Software-Hardware Interfaces. We have tailored the code security analysis approach to software close to the hardware level. More concretely, we considered the industrial Cryptographic Token Interface Standard PKCS 11 which defines how software on untrustworthy hardware can make use of tamper-proof hardware such as smart-cards to perform cryptographic operations on sensitive data. We developed an approach for automated security analysis with first-order logic theorem provers of crypto protocol implementations making use of this standard.

Analyzing Security Configurations. We have also performed research on linking the UMLsec approach with the automated analysis of security-critical configuration data. For example, our tools automatically checks SAP R/3 user permissions for security policy rules formulated as UML specifications [13]. Because of its modular architecture and its standardized interfaces, the tool can be adapted to check security constraints in other kinds of application software, such as firewalls or other access control configurations.

Industrial Applications of MBSE include:

Biometric Authentication. For a project with an industrial partner, MBSE was chosen to support the development of a biometric authentication system at the specification level, where three significant security flaws were found [14]. We also applied it to the source-code level for a prototypical implementation constructed from the specification [12].

Common Electronic Purse Specifications. MBSE was applied to a security analysis of the Common Electronic Purse Specifications (CEPS), a candidate for a globally interoperable electronic purse standard supported by organizations representing 90 % of the world's electronic purse cards (including Visa International). We found three significant security weaknesses in the purchase and load transaction protocols [13], proposed improvements to the specifications and showed that these are secure [13]. We also performed a security analysis of a prototypical Java Card implementation of CEPS.

Web-based Banking Application. In a project with a German bank, MBSE was applied to a web-based banking application to be used by customers to fill out and sign digital order forms [6]. The personal data in the forms must be kept confidential, and orders securely authenticated. The system uses a proprietary client authentication protocol layered over an SSL connection supposed to provide confidentiality and server authentication. Using the MBSE approach, the system architecture and the protocol were specified and verified with regard to the relevant security requirements.

In other applications [13], MBSE was used ...

- to uncover a flaw in a variant of the Internet protocol TLS proposed at IEEE Infocom 1999, and suggest and verify a correction of the protocol.
- to perform a security verification of the Java implementation Jessie of SSL.
- to correctly employ advanced Java 2 or CORBA security concepts in a way that allows an automated security analysis of the resulting systems.
- for an analysis of the security policies of a German mobile phone operator [17].
- for a security analysis of the specifications for the German Electronic Health Card in development by the German Ministry of Health.
- for the security analysis of an electronic purse system developed for the Oktoberfest in Munich.
- for a security analysis of an electronic signature pad based contract signing architecture under consideration by a German insurance company.

- in a project with a German car manufacturer for the security analysis of an intranet-based web information system.
- with a German chip manufacturer and a German reinsurance company for security risk assessment, also regarding Return on Security Investment.
- in applications specifically targeted to service-based, health telematics, and automotive systems.

Recently, there has been some work analyzing trade-offs between security- and performance-requirements [24,31].

3 Modelling Evolution with UMLseCh

This section introduces extensions of the UMLsec profile for supporting system evolution in the context of model-based secure software development with UML.

This profile, UMLseCh, is a further extension of the UML profile UMLsec in order to support system evolution in the context of model-based secure software development with UML. It is a "light-weight" extension of the UML in the sense that it is defined based on the UML notation using the extension mechanisms stereotypes, tags, and constraints, that are provided by the UML standard. For the purposes of this section, by "UML" we mean the core of the UML 2.0 which was conservatively included from UML 1.5[2].

As such, one can define the meta-model for UMLsec and also for UMLseCh by referring to the meta-model for UML and by defining the relevant list of stereotypes and associated tags and constraints. The meta-model of the UMLsec notation was defined in this way in [13]. In this section, we define the meta-model of UMLseCh in an analogous way.

The UMLseCh notation is divided in two parts: one part intended to be used during abstract design, which tends to be more informal and less complete in its use and is thus particularly suitable for abstract documentation and discussion with customers (cf. Sect. 3.1), and one part intended to be used during detailed design, which is assumed to be more detailed and also more formal, such that it will lend itself towards automated security analysis (cf. Sect. 3.2). We discuss about possible verification strategies based on the concrete notation in Sect. 3.3.

3.1 Abstract Notation

We use stereotypes to model change in the UML design models. These extend the existing UMLsec stereotypes and are specific for system evolution (change). We define change stereotypes on two abstraction layers: (i) abstract stereotypes and (ii) Concrete stereotypes. This subsection given an overview of the abstract stereotypes.

The aim of the abstract change stereotypes is to document change artefacts directly on the design models to enable controlled change actions. The abstract change stereotypes are tailored for modelling a living security system, i.e., through all phases of a system's life-cycle.

[2] http://www.omg.org/spec/UML/1.5

We distinguish between past, current and future change. The abstract stereotypes makes up three refinement levels, where the upper level is ≪`change`≫. This stereotype can be attached to subsystems and is used across all UML diagrams. The meaning of the stereotype is that the annotated modelling element and all its sub-elements has or is ready to undergo change.

≪`change`≫ is refined into the three change schedule stereotypes:

1. ≪`past_change`≫ representing changes already made to the system (typically between two system versions).
2. ≪`current_change`≫ representing changes currently being made to a system.
3. ≪`future_change`≫ specifying the future allowed changes.

To track and ensure controlled change actions one needs to be explicit about which model elements are allowed to change and what kind of change is permitted on a particular model element. For example, it should not be allowed to introduce audit on data elements that are private or otherwise sensitive, which is annotated using the UMLsec stereotype ≪`secrecy`≫. To avoid such conflict, security analysis must be undertaken by refining the abstract notation into the concrete one.

Past and current changes are categories into addition of new elements, modification of existing elements and deletion of elements. The following stereotypes have been defined to cover these three types of change: ≪`new`≫, ≪`modified`≫ and ≪`deleted`≫.

For future change we also include the same three categories of change and the following three future change stereotypes have been defined: ≪`allowed_add`≫; ≪`allowed_modify`≫; ≪`allowed_delete`≫. These stereotypes can be attached to any model element in a subsystem. The future change stereotypes are used to specify future allowed changes for a particular model element.

Past and current change The ≪`new`≫ stereotype is attached to a new system part that is added to the system as a result of a functionality-driven or a security-driven change. For security-driven changes, we use the UMLsec stereotypes secrecy, integrity and authenticity to specify the cause of security-driven change; e.g. that a component has been added to ensure the secrecy of information being transmitted. This piece of information allows us to keep track of the reasons behind a change. Such information is of particular importance for security analysis; e.g. to determine whether or which parts of a system (according to the associated dependencies tag) that must be analysed or added to the target of evaluation (ToE) in case of a security assurance evaluation.

Tagged values are used to assist in security analysis and holds information relevant for the associated stereotype. The tagged value: {`version=version_number`} is attached to the ≪`new`≫ stereotype to specify and trace the number of changes that has been made to the new system part. When a 'new' system part is first added to the system, the version number is set to 0. This means that if a system part has the ≪`new`≫ stereotype attached to it where the version number is > 0, the system part has been augmented with additional parts since being added to the system (e.g., addition of an new attribute to a new class). For all other changes, the ≪`modified`≫ stereotype shall be used.

The tagged value: {`dependencies=yes/no`} is used to document whether there is a dependency between other system parts and the new/modified system part. At this point in the work, we envision changes to this tag, maybe even a new stereotype to keep track of exactly which system parts that depends on each other. However, there is a need to gain more experience and to run through more examples to make a decision on this issue, as new stereotypes should only be added if necessary for the security analysis or for the security assurance evaluation. Note that the term dependencies are adopted from ISO 14508 Part 2 (Common Criteria) [5].

The ≪`modified`≫ change stereotype is attached to an already existing system part that has been modified as a result of a functional-driven or a security-driven change/change request. The tagged values is the same as for the 'new' stereotype.

The ≪`deleted`≫ change stereotype is attached to an existing system part (subsystem, package, node, class, components, etc.) for which one or more parts (component, attributes, service and similar) have been removed as a result of a functionality-driven change. This stereotype differs from the 'new' and 'modified' stereotypes in that it is only used in cases where it is essential to document the deletion. Examples of such cases are when a security component is removed as a result of a functionality-driven change, as this often affects the overall security level of a system. Information about deleted model elements are used as input to security analysis and security assurance evaluation.

Future change The allowed future change for a modelling element or system part (subsystem) is adding a new element, modifying an existing element and deleting elements (≪`allowed_add`≫, ≪`allowed_modify`≫ and ≪`allowed_delete`≫). We reuse the tagged values from the past and current change stereotypes, except for 'version_number' which is not used for future changes.

3.2 Concrete Notation

We further extend UMLsec by adding so called "concrete" stereotypes: these stereotypes allow to precisely define substitutive (sub) model elements and are equipped with constraints that help ensuring their correct application. These differ from the abstract stereotypes basically because we define a precise semantics (similar to the one of a transformation language) that is intended to be the basis for a security-preservation analysis based on the model difference between versions.

Figure 4 shows the stereotypes defining table. The tags table is shown in Figure 5.

Description of the notation. We now describe informally the intended semantics of each stereotype.

substitute. The stereotype substitute attached to a model element denotes the possibility for that model element to be substituted by a model element of the

Stereotype	Base Class	Tags	Constraints	Description
substitute	all	ref, substitute, pattern	FOL formula	substitute a model element
add	all	ref, add, pattern	FOL formula	add a model element
delete	all	ref, pattern	FOL formula	delete a model element
substitute-all	subsystem	ref, substitute, pattern	FOL formula	substitute a group of elements
add-all	subsystem	ref, add, pattern	FOL formula	add a group of elements
delete-all	subsystem	ref, pattern	FOL formula	delete a group of elements

Fig. 4. UMLsecCh concrete design stereotypes

Tag	Stereotype	Type	Multip.	Description
ref	substitute, add, delete, substitute-all, add-all, delete-all	object name	1	Informal type of change
substitute	substitute, substitute-all	list of model elements	1	Substitutives elements
add	add, add-all	list of model elements	1	New elements
pattern	substitute, add, delete, substitute-all, add-all, delete-all	list of model elements	1	Elements to be modified

Fig. 5. UMLsecCh concrete design tags

same type over the time. It has three associated tags, namely {`ref`}, {`substitute`} and {`pattern`}.

These tags are of the form

$$\{\,\mathsf{ref} = \mathsf{CHANGE\text{-}REFERENCE}\,\},$$
$$\{\,\mathsf{substitute} = \mathsf{MODEL\text{-}ELEMENT}\,\}$$
$$\text{and } \{\,\mathsf{pattern} = \mathsf{CONDITION}\,\}.$$

The tag {`ref`} takes a string as value, which is simply used as a reference of the change. The value of this tag can also be considered as a predicate and take a truth value to evaluate conditions on the changes, as we explain further in this section. The tag {`substitute`} has a list of model element as value, which represents the several different new model elements that can substitute the actual one if a change occurs. An element of the list contained in the tagged value is a model element itself (e.g. a stereotype, {`substitute = ≪stereotype≫`}). To textually describe UML model elements one can use an abstract syntax as in [13] or any equivalent grammar. Ideally, tool support should help the user into choosing from a graphical menu which model elements to use, without the user to learn the model-describing grammar. The last tag, {pattern}, is optional. If the model element to change is clearly identified by the syntactic notation, i.e. if there is no possible ambiguity to state which model element is concerned by the stereotype ≪substitute≫, the tag pattern can be omitted. On the other hand, if the model element concerned by the stereotype ≪substitute≫ is not clearly identifiable (as it will be the case for simultaneous changes where we can not attach the evolution stereotype to all targeted elements at once), the tag pattern must be used. This tag has a model element as value, which represents the model element to substitute if a change occurs. In general the value of pattern can be a function uniquely identifying one or more model elements within a diagram.

Therefore, to specify that we want to change, for example, a link stereotyped ≪Internet≫ with a link stereotyped ≪encrypted≫, using the UMLseCh notation, we attach:

≪substitute≫
{ ref = encrypt-link }
{ substitute = encrypted }
{ pattern = Internet }

to the link concerned by the change.

The stereotype ≪substitute≫ also has a constraint formulated in first order logic. This constraint is of the form [CONDITION]. As mentioned earlier, the value of the tag {ref} of a stereotype ≪substitute≫ can be used as the atomic predicate for the constraint of another stereotype ≪substitute≫. The truth value of that atomic predicate is true if the change represented by the stereotype ≪substitute≫ for which the tag {ref} is associated occurred, false otherwise. The truth value of the condition of a stereotype ≪substitute≫ then represents whether or not the change is allowed to happen (i.e. if the condition is evaluated to true, the change is allowed, otherwise the change is not allowed).

To illustrate the use of the constraint, let us refine the previous example. Assume that to allow the change with reference { ref = encrypt-link }, another change, simply named "change" for example, has to occur. We then attach the following to the link concerned by the change:

≪substitute≫
{ ref = encrypt-link }
{ substitute = encrypted }
{ pattern = Internet }
[change]

add. The stereotype ≪add≫ is similar to the stereotype ≪substitute≫ but, as its name indicates, denotes the addition of a new model element. It has three associated tags, namely {ref}, {add} and {pattern}. The tag {ref} has the same meaning as in the case of the stereotype ≪substitute≫, as well as the tag {add} (i.e. a list of model elements that we wish to add). The tag{pattern} has a slightly different meaning in this case. While with stereotype ≪`substitute`≫, the tag {`pattern`} represents the model element to substitute, within the stereotype ≪`add`≫ it does not represent the model element to add, but the parent model element to which the new (sub)-model element is to be added.

The stereotype ≪`add`≫ is a syntactic sugar of the stereotype ≪`substitute`≫, as a stereotype ≪`add`≫ could always be represented with a substitute stereotype (substituting the parent element with a modified one). For example, in the case of Class Diagrams, if s is the set of methods and m a new method to be added, the new set of methods is:

$$s' = s \cup \{m\}$$

The stereotype ≪`add`≫ also has a constraint formulated in first order logic, which represents the same information as for the stereotype ≪`substitute`≫.

delete. The stereotype ≪`delete`≫ is similar to the stereotype ≪`substitute`≫ but, obviously, denotes the deletion of a model element. It has two associated tags, namely {ref} and {pattern}, which have the same meaning as in the case of the stereotype ≪`substitute`≫, i.e. a reference name and the model element to delete respectively.

The stereotype ≪`delete`≫ is a syntactic sugar of the substitute stereotype, as a stereotype ≪`delete`≫ could always be represented with a substitution. For example, if s is the set of methods and m a method to delete, the new set of methods is:

$$s' = s \setminus m$$

As with the previous stereotypes, the stereotype ≪`delete`≫ also has a constraint formulated in first order logic.

substitute-all. The stereotype ≪`substitute-all`≫ is an extension of the stereotype ≪`substitute`≫ that can be associated to a (sub)model element or to a whole subsystem. It denotes the possibility for **a set of (sub)model elements** to evolve over the time and what are the possible changes. The elements of the set are sub elements of the element to which this stereotype is attached (i.e. a set of methods of a class, a set of links of a Deployment diagram, etc). As the stereotype ≪`substitute`≫, it has the three associated tags {`ref`}, {`substitute`} and {`pattern`}, of the form { ref = CHANGE-REFERENCE }, { substitute = MODEL-ELEMENT } and { pattern = CONDITION }. The tags {`ref`} and {`substitute`} have the exact same meaning as in the case of the stereotype ≪`substitute`≫. The tag {`pattern`}, here, does not represent one (sub)model element but **a set of (sub)model elements** to substitute if a change occur. Again, in order to identify the list model elements precisely, we can use, if necessary, the abstract syntax of UMLsec, defined in [13].

If we want, for example, to replace all the links stereotyped ≪`Internet`≫ of a subsystem by links stereotyped ≪`encrypted`≫, we can then attach the following to the subsystem:

$$\begin{gathered}\ll\texttt{substitute-all}\gg\\ \{\,\mathsf{ref}=\mathsf{encrypt\text{-}all\text{-}links}\,\}\\ \{\,\mathsf{substitute}=\ll\texttt{encrypted}\gg\,\}\\ \{\,\mathsf{pattern}=\ll\texttt{Internet}\gg\,\}\end{gathered}$$

The tags {substitute} and {pattern} here allow a parametrisation of the tagged values MODEL-ELEMENT and CONDITION in order to keep information of the different model elements of the subsystem concerned by the substitution. For this, we allow the use of variables in the tagged value of both, the tag {substitute} and the tag {pattern}.

To illustrate the use of the parametrisation in this simultaneous substitution, consider the following example. Assume that we would like to substitute all the secrecy tags in the stereotype ≪critical≫ by the integrity tag, we can attach:

$$\begin{gathered}\ll\texttt{substitute-all}\gg\\ \{\,\mathsf{ref}=\mathsf{secrecy\text{-}to\text{-}integrity}\,\}\\ \{\,\mathsf{substitute}=\{\,\mathsf{integrity}=\mathsf{X}\,\}\,\}\\ \{\,\mathsf{pattern}=\{\,\mathsf{secrecy}=\mathsf{X}\,\}\,\}\end{gathered}$$

to the model element to which the stereotype ≪critical≫ is attached.

The stereotype ≪substitute-all≫ also has a constraint formulated in first order logic, which represents the same information as for the stereotype ≪substitute≫.

add-all. The stereotype ≪add≫ also has its extension ≪add-all≫, which follows the same semantics as ≪substitue-all≫ but in the context of an addition.

delete-all. The stereotype ≪delete≫ also has its extension ≪delete-all≫, which follows the same semantics as ≪substitue-all≫ but in the context of a deletion.

Example. Figure 6 shows the use of ≪add-all≫ and ≪substitute-all≫ on a package containing a class diagram and a deployment diagram depicting the communication between two parties through a common proxy server. The change reflects the design choice to, in addition to protect the integrity of the message d, enforce the secrecy of this value as well.

Complex changes. In case of complex changes, that arise for example if we want to merge two diagrams having elements in common, we can overload the aforementioned stereotypes for accepting not only lists of elements but even lists of lists of elements. This is technically not very different from what we have described so far, since the complex evolutions can be seen as syntactic sugar for multiple coordinated single-model evolutions.

3.3 Security Preservation under Evolution

With the use of the UMLseCh concrete stereotypes, evolving a model means that we either *add*, *delete*, or / and *substitute* elements of this model explicitly.

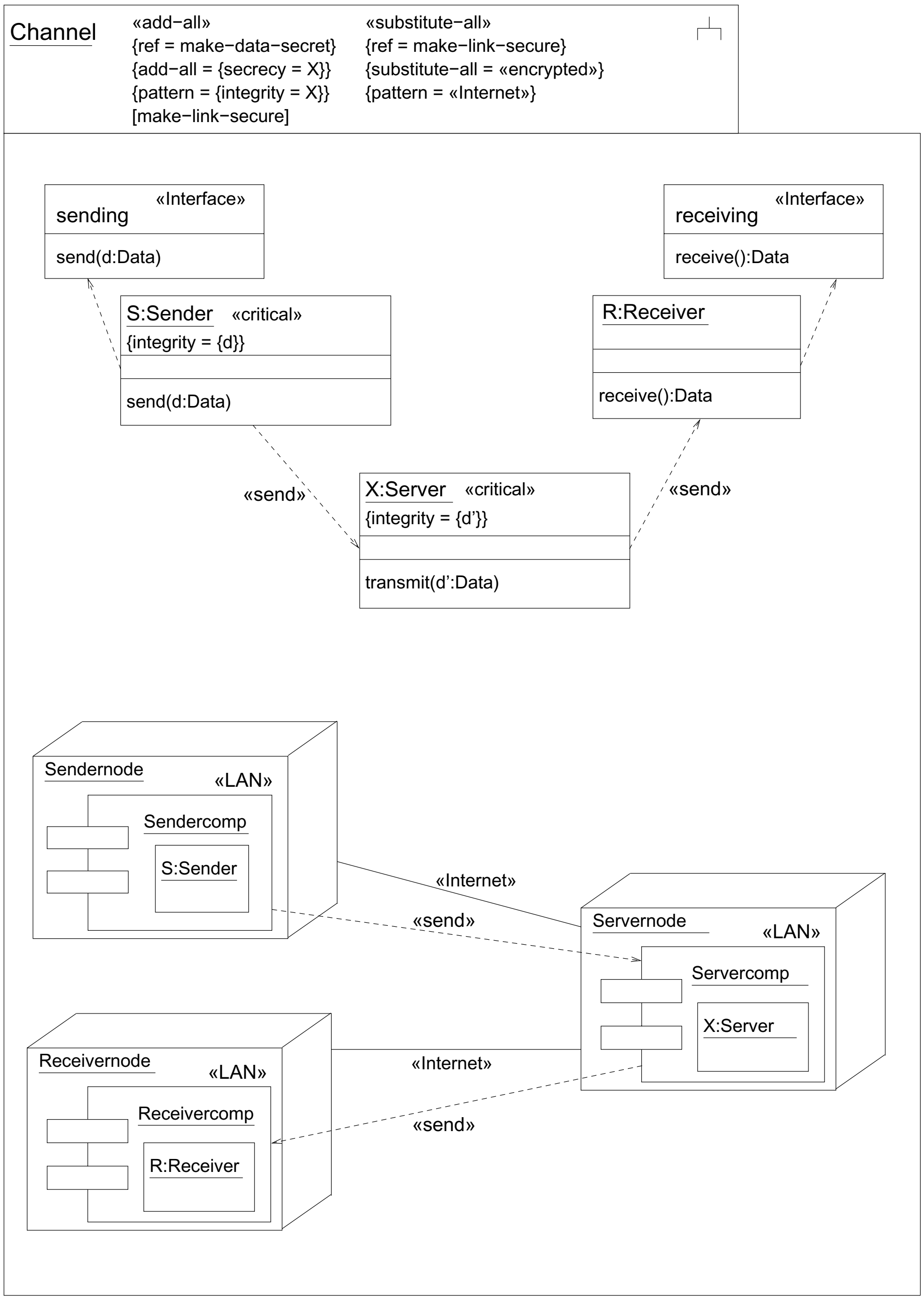


Fig. 6. A UMLseCh annotated diagram with simultaneous substitutions and additions

In other words, the stereotypes induce sets **Add**, **Del**, and **Subs**, containing the model elements to be added, deleted and substituted respectively, together with information about *where* to perform these changes.

Given a diagram M and a set Δ of such modifications we denote $M[\Delta]$ the diagram resulting after the modifications have taken place. So in general let P be a diagram property. We express the fact that M enforces P by $P(M)$. *Soundness* of the security preserving rules R for a property P on diagram M can be formalized as follows:

$$P(M) \wedge R(M, \Delta) \Rightarrow P(M[\Delta]).$$

So to reason about security preservation, one has several alternatives, depending on the property P. For some static analysis, it suffices to show that simultaneous sub-changes contained in Δ preserve P. Then, incrementally, we can proceed until reaching $P(M[\Delta])$. This can be done by reasoning inductively inductively by cases given a security requirement on UML models, by considering incremental atomic changes and distinguishing them according to a) their *evolution* type (addition, deletion, substitution) and b) their *UML diagram* type.

For dealing with behavioural properties one could exploit the divide and conquer approach by means of compositionality verification results. This idea, originally described in general in [4] (and as mentioned before, used for safety properties in [3]), is as follows: given components C and D, we denote with $C \otimes D$ its composition. Then, if we now that a security property P holds on both components separately and some set of rules R are satisfied then P holds in the composition, we can use this to achieve a more efficient verification under evolution, given R is easy enough to check. In practice, one often has to modify just one component in a system, and thus if one has:

$$P(C) \wedge P(D) \wedge R(C, D) \Rightarrow P(C \otimes D)$$

one can then check, for a modification Δ on one of the components:

$$P(C[\Delta]) \wedge P(D) \wedge R(C[\Delta], D) \Rightarrow P(C[\Delta] \otimes D) = P(C \otimes D)[\Delta]$$

and thus benefit from the already covered case $P(D)$ and the efficiency of R. Depending on the completeness of R, this procedure can also be relevant for an evolution of both components, since one could reapply the same decision procedure for a change in D (and therefore can be generalized to more than two components). The benefit consists in splitting the problem of verifying the composition (which is a problem with a bigger input) in two smaller subproblems. Some security properties (remarkably information flow properties like non-interference) are not safety properties, and there are interesting results for their compositionality (for example [23]).

4 Application Examples and Tool Support

4.1 Modelling Change of UMLsec Diagrams

Secure Dependency. This stereotype, used to label subsystems containing object diagrams or static structure diagrams, ensures that the ≪`call`≫ and ≪`send`≫

dependencies between objects or subsystems respect the security requirements on the data that may be communicated along them, as given by the tags {`secrecy`}, {`integrity`}, and {`high`} of the stereotype ≪`critical`≫. More exactly, the constraint enforced by this stereotype is that if there is a ≪`call`≫ or ≪`send`≫ dependency from an object (or subsystem) C to an interface I of an object (or subsystem) D then the following conditions are fulfilled.

– For any message name n in I, n appears in the tag {`secrecy`} (resp. {`integrity`}, {`high`}) in C if and only if it does so in D.
– If a message name in I appears in the tag {`secrecy`} (resp.{`integrity`}, {`high`}) in C then the dependency is stereotyped ≪`secrecy`≫ (resp. ≪`integrity`≫ , ≪`high`≫).

If the dependency goes directly to another object (or subsystem) without involving an interface, the same requirement applies to the trivial interface containing all messages of the server object.

This property is specially interesting to verify under evolution since it is local enough to re-use effectively previous verifications on the unmodified parts and its syntactic nature makes the incremental decision procedure relatively straightforward.

Example. The example in Fig. 7 shows the Client side of a communication channel between two parties. At first (disregarding the evolution stereotypes) the communication is unsecured. In the packages Symmetric and Asymmetric, we have classes providing cryptographic mechanisms to the Client class. Here the stereotype ≪add≫ marked with the reference tag {`ref`} with value add_encryption specifies two possible evolution paths: merging the classes contained in the current package (Channel) with either Symmetric or Asymmetric. There exists also a stereotype ≪add≫ associated with the Client class adding either a pre-shared private key k or a public key $\mathsf{K_S}$ of the server. To coordinate the intended evolution paths for these two stereotypes, we can use the following first-order logic constraint (associated with add_encryption):

$$\begin{aligned}[&\mathsf{add\_encryption(add) = Symmetric \Rightarrow add\_keys(add) = k : Keys} \ \wedge \\ &\mathsf{add\_encryption(add) = Asymmetric \Rightarrow add\_keys(add) = K_S : Keys}]\end{aligned}$$

The two deltas, representing two possible evolution paths induced by this notation, can be checked incrementally by case distinction. In this case, the evolution is security preserving in both cases. For more details about the verification technique see [27].

Role-based Access Control. The stereotype ≪`rbac`≫ defines the access rights of actors to activities within an activity diagram under a role schema. For this purpose there exists tags {protected}, {role}, {right}. An activity diagram is UMLsec satisfies ≪`rbac`≫ if for every protected activity A in {protected}, for which an user U has access to it, there exists a pair (A,R) in {rights} and a pair (R,U) in {roles}. The verification computational cost depends therefore on the number of protected activities.

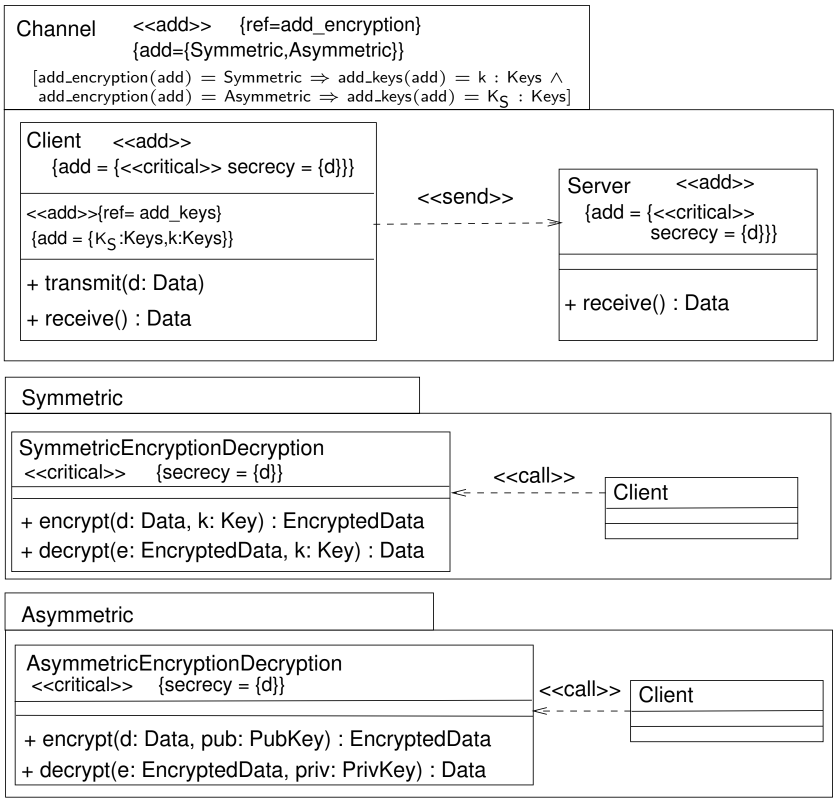

Channel <<add>> {ref=add_encryption}
{add={Symmetric,Asymmetric}}
[add_encryption(add) = Symmetric ⇒ add_keys(add) = k : Keys ∧
add_encryption(add) = Asymmetric ⇒ add_keys(add) = K_S : Keys]
Client <<add>>
{add = {<<critical>> secrecy = {d}}}
<<add>>{ref= add_keys}
{add = {K_S :Keys,k:Keys}}
+ transmit(d: Data)
+ receive() : Data
<<send>>
Server <<add>>
{add = {<<critical>> secrecy = {d}}}
+ receive() : Data
Symmetric
SymmetricEncryptionDecryption
<<critical>> {secrecy = {d}}
+ encrypt(d: Data, k: Key) : EncryptedData
+ decrypt(e: EncryptedData, k: Key) : Data
<<call>>
Client
Asymmetric
AsymmetricEncryptionDecryption
<<critical>> {secrecy = {d}}
+ encrypt(d: Data, pub: PubKey) : EncryptedData
+ decrypt(e: EncryptedData, priv: PrivKey) : Data
<<call>>
Client

Fig. 7. An evolving class diagram with two possible evolution paths

Example. In Fig. 8 (left-hand side), we show an activity diagram to which we want to add a new set of activities, introducing a web-check-in functionality to a flight booking system. The new activity "Check-in online" (middle of Fig. 8) is protected, but we do not add a proper role/right association to this activity, thus resulting in a security violating diagram (right-hand side Fig. 8).

4.2 Tool Support

The UMLsec Tool Suite provides mechanical tool support for analyzing UML specifications for security requirements using model-checkers and automated theorem provers for first-order logic . The tool support is based on an XML dialect called XMI which allows interchange of UML models. For this, the developer creates a model using a UML drawing tool capable of XMI export and stores it as an XMI file. The file is imported by the UMLsec analysis tool (for example, through its web interface) which analyses the UMLsec model with respect to the security requirements that are included. The results of the analysis are

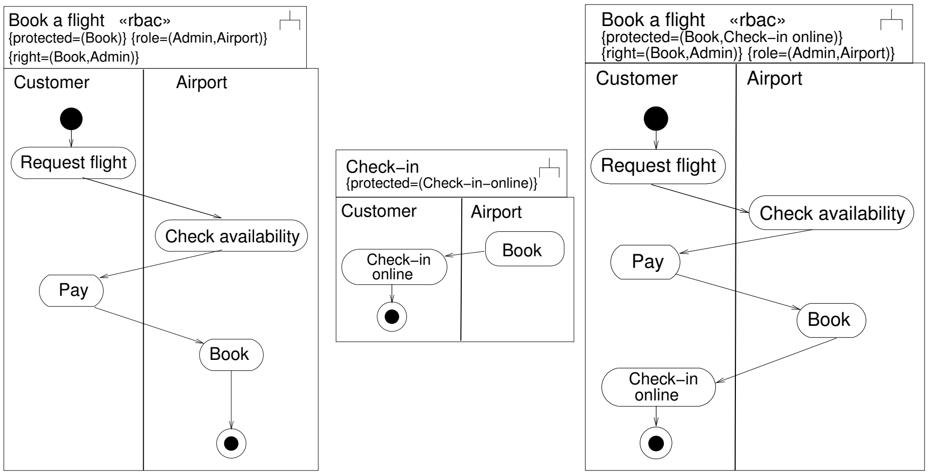


Fig. 8. Activity Diagram Annotated with ≪`rbac`≫ Before Evolution (left-hand side), Added Model Elements (middle), and After Evolution (right-hand side)

given back to the developer, together with a modified UML model, where the weaknesses that were found are highlighted.

We also have a framework for implementing verification routines for the constraints associated with the UMLsec stereotypes. The goal is that advanced users of the UMLsec approach should be able to use this framework to implement verification routines for the constraints of self-defined stereotypes. In particular, the framework includes the UMLsec tool web interface, so that new routines are also accessible over this interface. The idea behind the framework is to provide a common programming framework for the developers of different verification modules. A tool developer should be able to concentrate on the implementation of the verification logic and not be required to implement the user interface.

As mentioned in Sect. 3, tool support for UMLseCh would be beneficial in at least two ways:

– Supporting the user in modelling expected evolutions explicitly in a graphical way, without using a particular grammar or textual abstract syntax, and supporting the specification of non-elementary changes.
– Supporting the decision of including a change based on verification techniques for model consistency preservation after change.

but more importantly:

– Supporting the decision of including a change based on verification techniques for security preservation after change

First steps in this direction have been done in the context of the EU project SecureChange for statical security properties. For more details refer to [27].

5 Conclusions and Future Work

For system evolution to be reliable, it must be carried out in a controlled manner. Such control must include both functional and quality perspectives, such as security, of a system. Control can only be achieved under structured and formal identification and analysis of change implications up front, i.e. a priori. In this chapter, we presented a step-by-step controlled change process with emphasis on preservation of security properties through and throughout change, where the first is characterized as a security-driven change and the second a functionality-driven change. As there are many reasons for evolution to come about, security may drive the change process or be directly or indirectly influenced by it. Our approach covers both. In particular, the chapter introduces the change notation UMLseCh that allows for formally expressing, tracing, and analysing for security property preservation. UMLseCh can be viewed as an extension of UMLsec in the context of secure systems evolution. We showed how can one use the notation to model change for different UMLsec diagrams, and how this approach could be useful for tool-aided security verification. Consequently, this work can be extended in different directions. First of all, compositional and incremental techniques to reason about the security properties covered by UMLsec are necessary to take advantage of the precise model difference specification offered by UMLseCh. On the other hand, comprehensive tool support for both modelling and verification is key for a successful application of UMLseCh in practical contexts.

Acknowledgements

This research was partially supported by the EU project "Security Engineering for Lifelong Evolvable Systems" (*Secure Change*, ICT-FET-231101).

References

1. Andries, M., Engels, G., Habel, A., Hoffmann, B., Kreowski, H.-J., Kuske, S., Plump, D., Schürr, A., Taentzer, G.: Graph transformation for specification and programming. Science of Computer Programming 34(1), 1–54 (1999)
2. Bézivin, J., Büttner, F., Gogolla, M., Jouault, F., Kurtev, I., Lindow, A.: Model transformations? Transformation models! In: Wang, J., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 440–453. Springer, Heidelberg (2006)
3. Chaki, S., Sharygina, N., Sinha, N.: Verification of evolving software (2004)
4. Clarke, E., Long, D., McMillan, K.: Compositional model checking. In: Proceedings of the Annual Symposium on Logic in Computer Science (LICS), pp. 353–362 (June 1989)
5. ISO 15408:2007 Common Criteria for Information Technology Security Evaluation, Version 3.1, Revision 2: Part 2; Security Functional Components, CCMB-2007-09-002 (September 2007)

6. Grünbauer, J., Hollmann, H., Jürjens, J., Wimmel, G.: Modelling and verification of layered security protocols: A bank application. In: Anderson, S., Felici, M., Littlewood, B. (eds.) SAFECOMP 2003. LNCS, vol. 2788, pp. 116–129. Springer, Heidelberg (2003)
7. Heckel, R.: Compositional verification of reactive systems specified by graph transformation. In: Astesiano, E. (ed.) ETAPS 1998 and FASE 1998. LNCS, vol. 1382, pp. 138–153. Springer, Heidelberg (1998)
8. Höhn, S., Jürjens, J.: Rubacon: automated support for model-based compliance engineering. In: Robby [26], pp. 875–878
9. Jürjens, J.: Formal Semantics for Interacting UML subsystems. In: Proceedings of the International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS). International Federation for Information Processing (IFIP), pp. 29–44. Kluwer Academic Publishers, Dordrecht (2002)
10. Jürjens, J.: Principles for Secure Systems Design. PhD thesis, Oxford University Computing Laboratory (2002)
11. Jürjens, J.: Model-based security engineering with UML. In: Aldini, A., Gorrieri, R., Martinelli, F. (eds.) FOSAD 2005. LNCS, vol. 3655, pp. 42–77. Springer, Heidelberg (2005)
12. Jürjens, J.: Code security analysis of a biometric authentication system using automated theorem provers. In: Proceedings of the Annual Computer Security Applications Conference (ACSAC), pp. 138–149. IEEE Computer Society, Los Alamitos (2005)
13. Jürjens, J.: Secure Systems Development with UML. Springer, Heidelberg (2005)
14. Jürjens, J.: Sound methods and effective tools for model-based security engineering with UML. In: Roman, G.-C., Griswold, W.G., Nuseibeh, B. (eds.) Proceedings of the International Conference on Software Engineering (ICSE), pp. 322–331. ACM Press, New York (2005)
15. Jürjens, J.: Verification of low-level crypto-protocol implementations using automated theorem proving. In: MEMOCODE, pp. 89–98. IEEE, Los Alamitos (2005)
16. Jürjens, J.: Security analysis of crypto-based Java programs using automated theorem provers. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 167–176. IEEE Computer Society, Los Alamitos (2006)
17. Jürjens, J., Schreck, J., Bartmann, P.: Model-based security analysis for mobile communications. In: Robby [26], pp. 683–692
18. Jürjens, J., Shabalin, P.: Tools for secure systems development with UML. Intern. Journal on Software Tools for Technology Transfer 9(5-6), 527–544 (2007); Invited submission to the special issue for FASE 2004/05
19. Jürjens, J., Wimmel, G.: Formally testing fail-safety of electronic purse protocols. In: 16th International Conference on Automated Software Engineering (ASE 2001), pp. 408–411. IEEE Computer Society, Los Alamitos (2001)
20. Kolovos, D.S., Paige, R.F., Polack, F., Rose, L.M.: Update transformations in the small with the epsilon wizard language. Journal of Object Technology 6(9), 53–69 (2007)
21. Lehman, M.: Software's future: Managing evolution. IEEE Software 15(1), 40–44 (1998)
22. Lipson, H.: Evolutionary systems design: Recognizing changes in security and survivability risks. Technical Report CMU/SEI-2006-TN-027, Carnegie Mellon Software Engineering Institute (September 2006)

23. Mantel, H.: On the composition of secure systems. In: Proceedings of the IEEE Symposium on Security and Privacy, Oakland, CA, USA, pp. 88–101. IEEE Computer Society, Los Alamitos (2002)
24. Petriu, D.C., Woodside, C.M., Petriu, D.B., Xu, J., Israr, T., Georg, G., France, R.B., Bieman, J.M., Houmb, S.H., Jürjens, J.: Performance analysis of security aspects in UML models. In: Cortellessa, V., Uchitel, S., Yankelevich, D. (eds.) WOSP, pp. 91–102. ACM, New York (2007)
25. Rensink, A., Schmidt, Á., Varró, D.: Model checking graph transformations: A comparison of two approaches. In: Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G. (eds.) ICGT 2004. LNCS, vol. 3256, pp. 226–241. Springer, Heidelberg (2004)
26. Robby (ed.): 30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18. ACM, New York (2008)
27. Secure Change Project. Deliverable 4.2., `http://www-jj.cs.tu-dortmund.de/jj/deliverable_4_2.pdf`
28. UML Revision Task Force. OMG Unified Modeling Language: Specification. Object Management Group (OMG) (September 2001), `http://www.omg.org/spec/UML/1.4/PDF/index.htm`
29. UMLsec group. UMLsec Tool Suite (2001-2011), `http://www.umlsec.de`
30. Watson, B.: Non-functional analysis for UML models. In: Real-Time and Embedded Distributed Object Computing Workshop, Object Management Group (OMG), July 15-18 (2002)
31. Woodside, C.M., Petriu, D.C., Petriu, D.B., Xu, J., Israr, T.A., Georg, G., France, R.B., Bieman, J.M., Houmb, S.H., Jürjens, J.: Performance analysis of security aspects by weaving scenarios extracted from UML models. Journal of Systems and Software 82(1), 56–74 (2009)

Author Index