Luís Soares Barbosa   Markus Lumpe (Eds.)

Formal Aspects of Component Software

7th International Workshop, FACS 2010
Guimarães, Portugal, October 14-16, 2010
Revised Selected Papers

Springer

Volume Editors

Luís Soares Barbosa
Universidade do Minho
HASLab (High Assurance Software Laboratory)
and Dept. of Informatics
Campus de Gualtar, 4700-320 Braga, Portugal
E-mail: lsb@di.uminho.pt

Markus Lumpe
Swinburne University of Technology
Faculty of Information and Communication Technologies
P.O. Box 218, Hawthorn, VIC 3122, Australia
E-mail: mlumpe@ict.swin.edu.au

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-27268-4 e-ISBN 978-3-642-27269-1
DOI 10.1007/978-3-642-27269-1
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2011943630

CR Subject Classification (1998): D.2, F.3, D.3, D.2.4, F.4.1, D.1

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

On behalf of the Organizing Committee we are pleased to present the proceedings of the 7th International Workshop on Formal Aspects of Component Software (FACS 2010) organized by the University of Minho and held in Guimarães, Portugal during October 14–16, 2010.

The objective of FACS is to bring together researchers and practitioners in the areas of component software and formal methods in order to promote a deeper understanding of the component-based software development paradigm and its applications. The workshop seeks to develop a better understanding of how formal methods can or should be used to make component-based software development succeed. Formal methods consist of mathematically based techniques for the specification, development, and verification of software and hardware systems. They have shown their great utility in providing the formal foundations of component-based software and working out challenging issues such as mathematical models for components, composition and adaptation, or rigorous approaches to verification, deployment, testing, and certification.

FACS 2010 was the seventh event in a series of workshops, founded by the International Institute for Software Technology of the United Nations University (UNU-IIST). The first FACS workshop was co-located with FM 2003 (Pisa, Italy, September 2003). The following FACS workshops were organized as standalone events, at UNU-IIST in Macau (October 2005), at Charles University in Prague (September 2006), at INRIA in Sophia-Antipolis (September 2007), and at University of Málaga in Spain (September 2008). FACS 2009 was part of the Formal Methods Week in Eindhoven (October 2009).

The FACS 2010 program consisted of two keynotes given by Sanjit Seshia from the University of California, Berkeley, USA, and Luís Caires from the New University of Lisbon, Portugal, a panel discussion on service-oriented computing, and technical paper presentations (13 full papers and 6 Doctoral Track extended abstracts). The technical papers were carefully selected from a total of 37 submissions originating from 19 countries. Each paper was reviewed by at least three Program Committee members. The entire reviewing process was supported by the EasyChair Conference System. This LNCS volume contains the revised versions of the papers accepted for publication in the FACS 2010 proceedings.

We would like to express our gratitude to to all the researchers who submitted their work to the workshop and to all colleagues who served on the Program Committee and helped us prepare a high-quality workshop program. We are also grateful to the invited speakers, Sanjit Seshia from the University of California, Berkeley, USA, and Luís Caires from the New University of Lisbon, Portugal, for the willingness to present their research and perspectives on formal methods for component-based software at the workshop. And last but not least, we would like to thank the panel members Marjan Sirjan, Zhiming Liu, Carlos Canal, and Farhad Arbab for their valuable and inspiring contributions to a successful panel discussion that helped clarify the differences between the component-based and service-oriented paradigms.

FACS 2010 was financially supported by FCT, the Portuguese Foundation for Science and Technology, the School of Engineering of the University of Minho, and the CCTC Research Center. The active support of the International Institute for Software Technology of the United Nations University (UNU-IIST), at all stages of the workshop organization, is also deeply acknowledged.

A special word of gratitude is due to our colleagues at Minho who made this event possible: Sara Fernandes, Nuno Rodrigues, Nuno Oliveira and Hugo Macedo.

July 2011

Markus Lumpe
Luís S. Barbosa

Organization

Program Chairs

Luís S. Barbosa	Universidade do Minho, Portugal
Markus Lumpe	Swinburne University of Technology, Australia

Program Committee

Farhad Arbab	CWI, The Netherlands
Marco Autili	L'Aquila University, Italy
Luís S. Barbosa	Universidade do Minho, Portugal
Andreas Bauer	Australian National University, Australia
Frank S. de Boer	CWI, The Netherlands
Christiano Braga	Universidad Complutense de Madrid, Spain
Carlos Canal	Universidad de Málaga, Spain
Rolf Hennicker	LMU Munich, Germany
Einar Broch Johnsen	Universitetet i Oslo, Norway
Zhiming Liu	IIST UNU, Macau, China
Ying Liu	IBM China Research, China
Markus Lumpe	Swinburne University of Technology, Australia
Eric Madelaine	INRIA, Centre Sophia Antipolis, France
Sun Meng	CWI, The Netherlands
Corina Pasareanu	NASA Ames, USA
Patrizio Pelliccione	L'Aquila University, Italy
Frantisek Plasil	Charles University, Czech Republic
Anders Ravn	Aalborg University, Denmark
Nuno Rodrigues	IPCA, Portugal
Bernhard Schätz	Technical University of Munich, Germany
Marjan Sirjan	University of Tehran, Iran
Volker Stolz	UNU-IIST, Macau, China
Carolyn Talcott	SRI International, USA
Dang Van Hung	Vietnam National University, Vietnam
Naijun Zhan	IOS, China

Steering Committee

Zhiming Liu	IIST UNU, Macau, China, Coordinator
Farhad Arbab	CWI, The Netherlands
Luís S. Barbosa	Universidade do Minho, Portugal
Carlos Canal	University of Málaga, Spain
Markus Lumpe	Swinburne University of Technology, Australia

Eric Madelaine	INRIA, Sophia-Antipolis, France
Corina Pasareanu	NASA Ames Research Center, USA
Sun Meng	CWI, The Netherlands
Bernhard Schätz	Technical University of Munich, Germany

Local Organizing Committee

Hugo Macedo	Universidade do Minho, Portugal
Nuno Oliveira	Universidade do Minho, Portugal
Nuno Rodrigues	Polytechnic Institute of Cávado and Ave, Portugal
Sara Fernandes	Universidade do Minho, Portugal

External Referees

Ludwig Adam
Sebastian Bauer
Cristiano Bertolini
Jan Olaf Blech
Marcello M. Bonsangue
Michel Chaudron
Ludovic Henrio
Pavel Jezek
Narges Khakpour
Ehsan Khamespanah
Ramtin Khosravi
Martin Martin Schäf
Charles Morisset
Christian Pfaller
Tomas Poch
Tomas Pop
Hamideh Sabouri
Rudolf Schlatte
Ondrej Sery
Alexandra Silva
Silvia Lizeth Tapia Tarifa
Van Khanh To
Hoang Truong
Hieu Vo
Sebastian Voss
Shuling Wang
Ming Xu
Shaofa Yang
Liang Zhao

Sponsoring Institutions

FCT	- Science and Technology Foundation, Portugal
CCTC	- Centro de Ciências e Tecnologias de Computação, Portugal
EEUM	- School of Engineering of Minho University, Portugal
UNU-IIST	- International Institute of Software Technology, United Nations University, Macau, China

Table of Contents

Quantitative Analysis of Software: Challenges and Recent Advances

Sanjit A. Seshia

EECS Department, UC Berkeley
sseshia@eecs.berkeley.edu

Abstract. Even with impressive advances in formal methods over the last few decades, some problems in automatic verification remain challenging. Central amongst these is the verification of quantitative properties of software such as execution time or energy usage. This paper discusses the main challenges for quantitative analysis of software in cyber-physical systems. It also presents a new approach to this problem based on the combination of inductive inference with deductive reasoning. The approach has been implemented for timing analysis in a system called GAMETIME.

1 Introduction

Cyber-physical systems tightly integrate computation with the physical world. Consequently, the behavior of software controllers of such systems has a major effect on physical properties of such systems. These properties are *quantitative*, encoding specifications on physical quantities such as time, energy, position, and acceleration. The verification of such quantitative properties of cyber-physical software systems requires modeling not only the software program but also the relevant aspects of the program's environment. In contrast with traditional "Boolean" verification of software, environment models must be more precise — for example, one cannot liberally employ nondeterminism in modeling the environment, and one cannot abstract away the hardware or the network. This challenge of accurate modeling is one of the major reasons why the progress on quantitative software verification has lagged behind that on Boolean software verification.

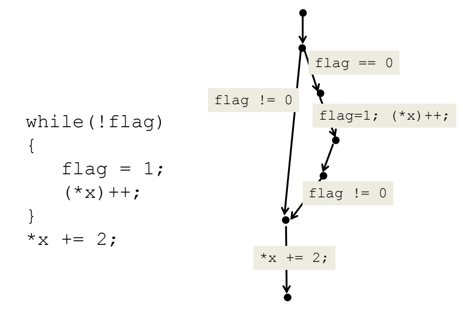
Consider, for example, the area of timing analysis of software. Several kinds of timing analysis problems arise in practice. First, for hard real-time systems, a classic problem is to estimate the worst-case execution time (WCET) of a terminating software task. Such an estimate is relevant for verifying if deadlines or timing constraints are met as well as for use in scheduling strategies. Second, for soft real-time systems, it can be useful to estimate the distribution of execution times exhibitable by a task. Third, it can be very useful to find a test case on which the program exhibits anomalous timing behavior; e.g., a test case causing a task to miss its deadline. Finally, in "software-in-the-loop" simulation, the software implementation of a controller is simulated along with a model of the continuous plant it controls, with the simulations connected using execution time estimates. For scalability, such simulation must be performed on a workstation, not on the target embedded platform. Consequently, during the workstation-based simulation, it is necessary to predict the timing of the program along a particular execution path on the target platform. All of these problems are instances of *predicting* a particular execution time property of a terminating software task.

In particular, the problem of WCET estimation has been the subject of significant research efforts over the last 20 years (e.g. [3,4]). Significant progress has been made on this problem, especially in the computation of bounds on loops in tasks, in modeling the dependencies amongst program fragments using (linear) constraints, and modeling some aspects of processor behavior. However, as pointed out in recent papers (e.g., Lee [2]), it is becoming increasingly difficult to precisely model the complexities of the underlying hardware platform (e.g., out-of-order processors with deep pipelines, branch prediction, multi-level caches, parallelism) as well as the software environment. This results in timing estimates that are either too pessimistic (due to conservative platform modeling) or too optimistic (due to unmodeled features of the platform). Due to the difficulty of platform modeling, industry practice typically involves making random, unguided measurements to obtain timing estimates, but these provide no guarantees.

In Section 2, we elaborate on the challenge of *environment modeling*. Techniques for automatically inferring an adequate environment model are required to address this challenge. In Section 3, we present the first steps towards such solution, implemented in the timing analysis tool GAMETIME. We conclude in Section 4 with directions for future research.

2 Challenge: Environment Modeling

We discuss the challenge of environment modeling using timing analysis as an example. The complexity of the timing analysis arises from two dimensions of the problem: the *path dimension*, where one must find the worst-case computation path for the task, and the *state dimension*, where one must find the right (starting) environment state to run the task from. Moreover, these two dimensions interact closely; for example, the choice of path can affect the impact of the starting environment state.



```
while(!flag)
{
   flag = 1;
   (*x)++;
}
*x += 2;
```

Fig. 1. Simple Illustrative Example

Consider the toy C program in Fig. 2(a). It contains a loop, which executes at most once. Thus, the control-flow graph (CFG) of the program can be unrolled into a directed acyclic graph (DAG), as shown in Fig. 2(b). Suppose we execute this program on a simple processor with an in-order pipeline and a data cache. Consider executing this program from the state where the cache is empty. The final statement of the program, `*x += 2`, contains a load, a store, and an arithmetic operation. If the left-hand path is taken, the load will suffer a cache miss; however, if the right-hand path is taken, there is a cache hit. The difference in timing between a cache hit and a miss can be an order of magnitude. Thus, the time taken by this statement depends on the program path taken. However, if the program were executed from a state with the data in the cache, there will be a cache hit even if the left-hand path is taken.

Thus, even with this toy program and a simple processor, one can observe that a timing analysis tool must explore the space of all possible program paths – a potentially exponentially large search space. If the starting environment state is known, a compositional timing tool that seeks to predict path timing by measuring timing of basic blocks must (i) model the platform precisely to predict timing at basic block boundaries, and (ii) search an exponentially-large environment state space at these boundaries. If the starting environment state is unknown, the problem is even harder.

Current state-of-the-art tools for timing analysis rely heavily on manually-constructed abstract timing models to achieve the right balance of precision and scalability. Manual modeling can be extremely tedious and error-prone, requiring several man-months of effort to design a timing model correctly even for a simple microcontroller. The challenge of keeping up with today's advances in microarchitecture are really daunting.

3 Automatic Model Inference: The GameTime Approach

Automatic inductive inference of models offers a way to mitigate the challenge of environment modeling. In this approach, a *program-specific timing model* of the platform is inferred from carefully chosen observations of the program's timing. The program-specificity is an important difference from traditional approaches, which seek to manually construct a timing model that works for *all* programs one might run on the platform. The latter is a very hard problem and perhaps not one we need to necessarily solve! An accurate model for the programs of interest should suffice.

This approach has been implemented for timing analysis in a toolkit called GameTime [7,6,5]. In GameTime, the platform is viewed as an adversary that controls the choice and evolution of the environment state, while the tool has control of the program path space. The analysis problem is then a game between the tool and the platform. In contrast with most existing tools for timing analysis (see, e.g., [4]), GameTime can predict not only extreme-case behavior, but also certain execution time statistics (e.g., the distribution) as well as a program's timing along particular execution paths. Additionally, it only requires one to run end-to-end measurements on the target platform, making it easy to port to new platforms. The GameTime approach, along with an exposition of theoretical and experimental results, including comparisons with other methods, is described in existing papers [7,6,5]. We provide only a brief overview of the approach taken by GameTime here.

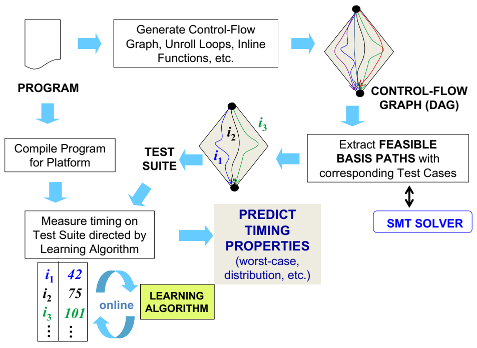

PROGRAM
Generate Control-Flow Graph, Unroll Loops, Inline Functions, etc.
CONTROL-FLOW GRAPH (DAG)
Compile Program for Platform
TEST SUITE
i_1
i_2
i_3
Extract FEASIBLE BASIS PATHS with corresponding Test Cases
SMT SOLVER
Measure timing on Test Suite directed by Learning Algorithm
PREDICT TIMING PROPERTIES (worst-case, distribution, etc.)
i_1 42
i_2 75
i_3 101
online
LEARNING ALGORITHM

Fig. 2. GameTime **overview**

Figure 2 depicts the operation of GameTime. As shown in the top-left corner, the process begins with the generation of the control-flow graph (CFG) corresponding to the program, where all loops have been unrolled to a maximum iteration bound, and all function calls have been inlined into the top-level function. The CFG is assumed to have a single source node (entry point) and a single sink node (exit point); if not, dummy source and sink nodes are added. The next step is a critical one, where a subset of program paths, called *basis paths* are extracted. These basis paths are those that form a basis for the set of all paths, in the standard linear algebra sense of a basis. A *satisfiability modulo theories (SMT) solver* — a deductive engine — is invoked to ensure that the generated basis paths are feasible. For each feasible basis path generated, the SMT solver generates a test case that drives program execution down that path. Thus a set of *feasible basis paths* is generated that spans the entire space of feasible program paths, along with the corresponding test cases.

The program is then compiled for the target platform, and executed on these test cases. In the basic GAMETIME algorithm (described in [7,6]), the sequence of tests is randomized, with basis paths being chosen uniformly at random to be executed. The overall execution time of the program is recorded for each test case. From these end-to-end execution time measurements, GAMETIME's learning algorithm generates a weighted graph model that is used to make predictions about timing properties of interest. The predictions hold with high probability under certain assumptions; see the previous papers on GAMETIME [7,6] for details. Experimental results indicate that in practice GAMETIME can accurately predict not only the worst-case path (and thus WCET) but also the distribution of execution times of a task from various starting environment states.

4 Looking Ahead

GAMETIME is only a first step and much remains to be done in the area of quantitative verification of software. First, the GAMETIME approach can be further refined to strengthen the theoretical guarantees and improve scalability, e.g., through compositional reasoning. Second, we need to understand what kind of support from platform designers (e.g., [1]) can make the quantitative verification problem easier. Third, many other quantitative verification problems, such as verifying bounds on energy consumption, remain to be fully explored. Finally, some of the fundamental ideas behind GAMETIME, such as the generation of feasible basis paths and the combination of inductive and deductive reasoning apply more generally beyond the area of cyber-physical systems.

Acknowledgments. This work was supported in part by NSF grants CNS-0644436, CNS-0627734, and CNS-1035672, an Alfred P. Sloan Research Fellowship, and the Multiscale Systems Center (MuSyC), one of six research centers funded under the Focus Center Research Program (FCRP), a Semiconductor Research Corporation entity.

References

1. Edwards, S.A., Lee, E.A.: The case for the precision timed (PRET) machine. In: Design Automation Conference (DAC), pp. 264–265 (2007)
2. Lee, E.A.: Computing foundations and practice for cyber-physical systems: A preliminary report. Technical Report UCB/EECS-2007-72, UC Berkeley (May 2007)

3. Li, Y.-T.S., Malik, S.: Performance Analysis of Real-Time Embedded Software. Kluwer Academic, Dordrecht (1999)
4. Wilhelm, R., et al.: The Determination of Worst-Case Execution Times—Overview of the Methods and Survey of Tools. In: ACM Transactions on Embedded Computing Systems, TECS (2007)
5. Seshia, S.A., Kotker, J.: GameTime: A toolkit for timing analysis of software. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 388–392. Springer, Heidelberg (2011)
6. Seshia, S.A., Rakhlin, A.: Quantitative analysis of systems using game-theoretic learning. ACM Transactions on Embedded Computing Systems (TECS) (to appear)
7. Seshia, S.A., Rakhlin, A.: Game-theoretic timing analysis. In: Proc. IEEE/ACM International Conference on Computer-Aided Design (ICCAD), pp. 575–582 (2008)

Analysis of Service Oriented Software Systems with the Conversation Calculus

Luís Caires and Hugo Torres Vieira

CITI and Departamento de Informática, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa

Abstract. We overview some perspectives on the concept of service-based computing, and discuss the motivation of a small set of modeling abstractions for expressing and analyzing service based systems, which have led to the design of the Conversation Calculus. Distinguishing aspects of the Conversation Calculus are the adoption of a very simple, context sensitive, local message-passing communication mechanism, natural support for modeling multi-party conversations, and a novel mechanism for handling exceptional behavior. In this paper, written in a tutorial style, we review some Conversation Calculus based analysis techniques for reasoning about properties of service-based systems, mainly by going through a sequence of illustrating examples.

1 Introduction

Web and service-based systems have emerged mainly as a toolkit of technological and methodological solutions for building open-ended collaborative applications on the internet, leading to the recent trend towards the SaaS (Software as a Service) distribution model. Many concepts frequently advanced as particular to service-oriented computing systems, namely, interface-oriented distributed programming, long duration transactions and compensations, separation of workflow from service instances, late binding and discovery of functionalities, are not new. However, it must be acknowledged that the idea of service based computing is definitely contributing to physically realize an emerging model of computation, which is global (encompassing the internet as a whole), interaction-based (subsystems communicate via message passing), and loosely coupled (connections are established dynamically, and on demand). It is then very important to better understand in what sense service orientation may be exploited as a new paradigm to build and reason about distributed systems.

Of course, the global computing infrastructure is bound to remain highly heterogeneous and dynamic, so it does not seem reasonable to foresee the premature emergence of comprehensive theories and technological artifacts, well suited for everyone and every application. This suggests that one should focus not only on particular systems and theories themselves, but also on general systems, their properties, and their interfaces. In a recent line of work, mainly developed in the context of the EU IP project Sensoria [21] and extended in the context of the

CMU-PT INTERFACES project [15], we have proposed a new model for service-oriented computation, based on a process calculus, with the aim of providing a foundation for rigorous modeling, analysis and verification. Our starting point was an attempt to isolate the essential characteristics of the service-oriented model of computation, in order to propose a motivation from "first principles" of a reduced set of general abstractions for expressing and analyzing service based systems. To focus on a set of independent primitives, we have developed our model by modularly adapting the synchronous π-calculus as follows

– introducing the general notion of *conversation context*;
– replacing channel communication by labeled message-passing primitives;
– adding a canonical exception handling mechanism

We have striven to keep our realization fairly general, so to achieve simplicity and clarity, and to ensure orthogonality and semantic independence of the chosen primitives. The proposed model, the Conversation Calculus, is a minimalistic, yet very expressive formalism, able to express and support reasoning about complex, dynamic, multiparty service based conversations, where partners may dynamically join and leave interactions, as we often find in"real-world" service based systems.

In this paper, we show how the Conversation Calculus can be used to reason about properties of service-based systems. In the spirit of a tutorial, no really new concepts are introduced, instead the focus is on a more detailed discussion of the underlying principles that guided the development of the language, accompanied by new examples that illustrate its expressiveness, and the kind of analyses that may be performed in the framework.

2 Aspects of Services

In this section, we attempt to identify some essential characteristics of the service-oriented model of computation, in order to justify a motivation from "first principles" of a reduced set of general abstractions for expressing and analyzing service based systems. Following [26], we identify as main features *distribution*, communication and *context* sensitiveness, and *loose coupling.*

2.1 Distribution

The purpose of a service relationship is to allow the incorporation of extra activities in a software system, without having to engage *local* resources and capabilities to support such activities. By delegating activities to an external provider, which will perform them using their remote resources and capabilities, a computing system may concentrate on those tasks for which it may autonomously provide adequate solutions. Thus, the notion of service makes particular sense when the service provider and the service client are separate entities, with access to separate resources and capabilities. This notion of the service relationship between provider and client assumes an underlying distributed computational model, where client and server are located at least in distinct (operating system) processes, more frequently in distinct network sites.

The invocation of a service by a client results in the creation of a new service instance. Initially, a service instance is composed by a pair of endpoints, one endpoint located in the server site, where the service is defined, the other endpoint in the client site, where the request for instantiation took place. From the viewpoint of each partner, the respective endpoint acts as a local process, with potential direct access to local resources and capabilities. Thus, for us an endpoint is not a name, a port address, or channel, but an interactive process. Endpoints work together in a tightly coordinated way, by exchanging data and control information through a dedicated communication medium. In general, a service relationship may be initiated by a pair of remote endpoints, but may later on be enlarged with extra endpoints, located in other sites, developing a multiparty conversation.

2.2 Communication, Contexts, and Context Sensitiveness

The invocation of a service by an initiator client causes the creation of a new communication medium to host the interactions of the particular service instance. The service client (initiator) and client (responder) are immediately given access to this freshly created communication medium (see Figure 1), which will host a new conversation, and which might later on be joined in by other parties. To interact in the conversation medium the client establishes an endpoint or access point to the medium in its local context, and the process located in the endpoint is able to interact remotely in the service medium and locally in the client context. Likewise, the service responder establishes an endpoint to interact in the service conversation, and the process located in the endpoint is able to communicate with the server context (e.g., to access server resources).

For example, consider the scenario where the endpoint realizing an archiving functionality in the client context communicates with the other subsystems of the client, e.g., to receive document archiving requests and document retrieval requests, while the remote endpoint in the server site communicates with other subsystems in the service provider context, e.g., the database, the indexing infrastructure, and other resources needed for the provider task.

Access to a conversation may be shared with other parties. In particular, access may be given to other service providers that may then contribute to ongoing service tasks. At any moment, any party may then interact with any other party that shares access to the same conversation, as depicted in Figure 2 for the case of three parties. It is important to notice that in general, the distinction client/server may get blurred in a multiparty conversation, and we essentially need to host a delimited conversation between several symmetric partners.

For instance, in the archiving example, the server may decide to share its work load with other providers, allowing each one to establish an endpoint of the shared medium. In such way, a request to store data may be picked up by any of the service providers listening on the shared communication medium.

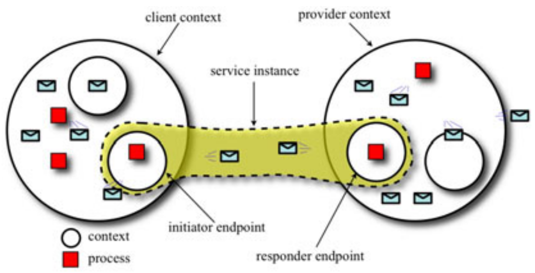


Fig. 1. Conversation initiation

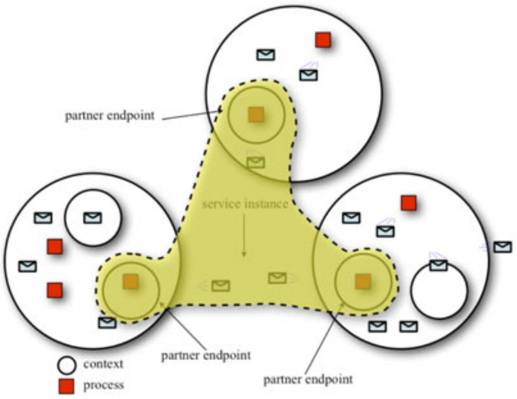


Fig. 2. Ongoing Conversation

We understand an endpoint just as a particular case of a delimited context or scope of interaction. More generally, a context is a delimited space were computation and communication happens. A context may have a spatial meaning, e.g., as a *site* in a distributed system, but also a behavioral meaning, e.g., as *context of conversation*, or a *session*, between two or more partners. For example, the same message may appear in two different contexts, with different meanings (web services technology has introduced artifacts such as “correlation” to determine the appropriate context for otherwise indistinguishable messages).

Thus, the notion of conversation as a medium of communication which may be accessed from anywhere in the system and shared between several parties seems to be a convenient abstraction mechanism to structure the several service interactions in a service-oriented system.

The description above suggests two forms of communication capabilities. First, processes may interact if they are located in the same endpoint or in two endpoints of the same conversation. Second, interaction may occur between immediately nested endpoints. Endpoints as the one described may be nested at many levels, corresponding to subsidiary service instances, processes, etc. Notice that we do not expect communication to happen between arbitrary contexts, but

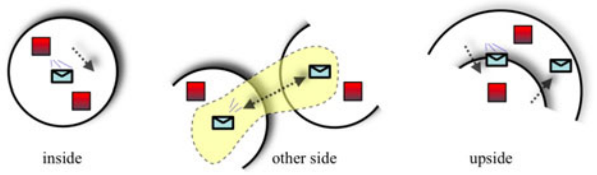


Fig. 3. Contexts and Communication Pathways

rather to always fall in one of the two special cases described above: interaction inside a given conversation, and external interaction (with the immediately external context). In Figure 3 we illustrate our intended context dependent communication model, and the various forms of interaction it admits.

A context is also a natural abstraction to group and publish together closely related services, including when such services are provided by several sites. Typically, services published by the same entity are expected to share common resources; we notice that such sharing is common at several scales of granularity. Extreme examples are an object, where the service definitions are the methods and the shared context is the object internal state, and an entity such as, e.g., Amazon, that publishes several services for many different purposes; such services certainly share many internal resources of the Amazon context, such as databases, payment gateways, and so on.

Delimited contexts are also natural candidates for typing, in terms of the messages interchange patterns that may happen at its border. We would thus expect types (or logical formulas) specifying various properties of interfaces, of service contracts, of endpoint session protocols, of security policies, of resource usage, and of service level agreements, to be in general assigned to context boundaries. Enforcing boundaries between subsystems is also instrumental to achieve loose coupling of systems.

2.3 Loose Coupling

A service task may rely on several subsidiary services, where each one may involve a number of collaborating partners, and some local processes that control (orchestrate) the several subsidiary tasks and carry out some local functionality. Crucially to the service-oriented design, the several pieces that form the service implementation should be composed in a loosely coupled way, so as to support aimed features of service-oriented systems such as dynamic binding and dynamic discovery of partner service providers. For instance, an orchestration describing a "business process", should be specified in a quite independent way of the particular subsidiary service instances used. In the orchestration language `WSBPEL` [2], loose coupling to external services is enforced to some extent by the separate declaration of "partner links" and "partner roles" in processes. In the modeling language SRML [18] (inspired by the Service Component Architecture [4]), the binding between service providers and clients is mediated by "wires", which describe plugging constraints between otherwise hard to match interfaces.

To support loose coupling, the specific details of a given service implementation should not be visible to external processes, so there must be a boundary between service instances and processes using them. Such boundary may be imposed by mediating processes that adapt the (implementation specific) service communication protocols to the abstract behavioral interface expected by the external context. It is then instrumental to encapsulate all computational entities cooperating in a service task in a conversation context, and allow them to communicate between themselves and the outer context only via some general message passing mechanism.

2.4 Other Aspects

There are many other aspects that must be addressed in a general model of service-oriented computation. The most obvious ones include failure handling and resource disposal, security (in particular access control, authentication and secrecy), time awareness, and a clean mechanism of inter-operation. This last aspect seems particularly relevant, and certainly suggests an important evaluation criteria for any service-oriented computation model.

3 The Conversation Calculus

In this section, we motivate and present in detail the primitives of our calculus. After that, we present the syntax of our calculus, and formally define its operational semantics, by means of a labeled transition system.

Conversation Context. A conversation context is a medium where related interactions can take place. A conversation context can be distributed in many pieces, and processes inside any piece can seamlessly talk to any other piece of the same context. Each context has a unique name (cf., a URI). We use the conversation access construct

$$n \blacktriangleleft [P]$$

to say that the process P is interacting in conversation n. Potentially, each conversation access will be placed at a different enclosing context. On the other hand, any such conversation access will necessarily be placed at a single enclosing context. The relationship between the enclosing context and such an access point may be seen as a caller/callee relationship, but where both entities may interact continuously.

Context Awareness. A process running inside a given context should be able to dynamically become aware of the identity of the former. This capability may be realized by the construct

$$\mathtt{this}(x).P$$

The variable x will be replaced inside the process P by the name n of the current context. The computation will proceed as $P\{x{\leftarrow}n\}$. For instance the process $c \blacktriangleleft [\mathtt{this}(x).P]$ evolves in one computation step to $c \blacktriangleleft [P\{x{\leftarrow}c\}]$. Our context awareness primitive bears some similarity with the `self` or `this` of object-oriented languages, although of course it has a different semantics.

3.1 Communication

Communication between subsystems is realized by message passing. We denote the input/output of messages from/to the current context by the constructs

$$\mathtt{label}^{\downarrow}?(x_1,\ldots,x_n).P$$
$$\mathtt{label}^{\downarrow}!(v_1,\ldots,v_n).P$$

Messages are `label`ed. In the output case (!), the terms v_i represent message arguments, that is, values to be sent, as expected. In the input case (?), the variables x_i represent the message parameters and are bound in P, as expected. The target symbol $\downarrow$ (read "here") says that the corresponding communication actions must interact in the current conversation conversation context, where the messages are being sent. Second, we denote the input/output of messages from/to the outer context by the constructs

$$\mathtt{label}^{\uparrow}?(x_1,\ldots,x_n).P$$
$$\mathtt{label}^{\uparrow}!(v_1,\ldots,v_n).P$$

The target symbol $\uparrow$ (read "up") says that the corresponding communication actions must interact in the (uniquely determined) outer context, where "outer" is understood relatively to the context where the process exercising the action is running.

3.2 Service Oriented Idioms

Although we do not introduce them natively in the language, we present some service-oriented idioms which capture typical service-oriented interaction: service definition, service instantiation and service join.

A context (a.k.a. a site) may publish one or more service definitions. Service definitions are stateless entities, pretty much as function definitions in a functional programming language. A service definition may be expressed by

$$\mathtt{def}\ \mathit{ServiceName} \Rightarrow \mathit{ServiceBody}$$

where *ServiceName* is the service name, and *ServiceBody* is the process that should be executed at the service endpoint for each service instance, in other words the service body. In order to be published, such a definition must be inserted into a context, e.g.,

$$\mathit{ServiceProvider} \blacktriangleleft [\mathtt{def}\ \mathit{ServiceName} \Rightarrow \mathit{ServiceBody}\ \cdots]$$

Such a published service may be instantiated by means of a instantiation idiom

$$\texttt{new}\ n \cdot \mathit{ServiceName} \Leftarrow \mathit{ClientProtocol}$$

where n identifies the conversation where the service is published. For instance, the service defined above may be instantiated by

$$\texttt{new}\ \mathit{ServiceProvider} \cdot \mathit{ServiceName} \Leftarrow \mathit{ClientProtocol}$$

The *ClientProtocol* describes the process that will run inside the endpoint held by the client. The outcome of a service instantiation is the creation of a new globally fresh context identity (a hidden name), and the creation of two access pieces of a context named by this fresh identity. One will contain the *ServiceBody* process and will be located inside the *ServiceProvider* context. The other will contain the *ClientProtocol* process and will be located in the same context as the `instance` expression that requested the service instantiation.

In our model conversation identifiers may be manipulated by processes if needed (accessed via the $\texttt{this}(x).P$), passed around in messages and subject to scope extrusion: this allows us to model collaboration between multiple parties in a single service conversation, realized by the progressive access of dynamically determined partners to an ongoing conversation. Joining of another partner to an ongoing conversation is a frequent programming idiom, that may be abstracted by:

$$\texttt{join}\ \mathit{ServiceProvider} \cdot \texttt{ServiceName} \Leftarrow \mathit{ContinuationProcess}$$

The `join` and the `new` expression are implemented in a similar way, both relying on name passing. The key difference is that while `new` creates a fresh *new conversation*, `join` allows a service `ServiceName` defined at *ServiceProvider* to join in the *current conversation*, while the calling party continues interacting in the current conversation as specified by *ContinuationProcess*. So, even if the `new` and `join` are represented in a similar way, the abstract notion they realize is actually very different: `new` is used to start a fresh conversation between two parties (e.g., used by a client that instantiates a service) while the `join` is used to allow another service provider to join an ongoing conversation (e.g., used by a participant in a service collaboration to dynamically delegate a task to some remote partner). At a very high level of description the two primitives can be unified as primitives that support the dynamic delegation of tasks (either in a unary conversation or in a n-ary conversation).

3.3 Exception Handling

We introduce two primitives to model exceptional behavior, in particular fault signaling, fault detection, and resource disposal, these aspects are certainly orthogonal to the previously introduced communication mechanisms. We adapt the classical $\texttt{try} - \texttt{catch}-$ and $\texttt{throw}-$ to a concurrent setting. The primitive to raise an exception is

$$\texttt{throw}.\mathit{Exception}$$

This construct throws an exception with continuation the process *Exception*, and has the effect of forcing the termination of all other processes running in all enclosing contexts, up to the point where a `try` – `catch` block is found (if any). The continuation *Exception* will be activated when (and if) the exception is caught by such an exception handler. The exception handler construct

$$\texttt{try}\ P\ \texttt{catch}\ \mathit{Handler}$$

allows a process P to execute normally until some exception is thrown inside P. At that moment, all of P is terminated, and the *Handler* handler process, which is guarded by `try` – `catch`, is activated, concurrently with the continuation *Exception* of the $\texttt{throw}.\mathit{Exception}$ that originated the exception, in the context of a given `try` – `catch`– block. By exploiting the interaction of the *Handler* and *Exception* processes, it is possible to represent many recovery and resource disposal protocols, including compensable transactions [12].

3.4 Syntax and Semantics of the Calculus

We may now formally introduce the syntax and semantics of the conversation calculus. We assume given an infinite set of names Λ, an infinite set of variables $\mathcal{V}$, and an infinite set of labels $\mathcal{L}$. We abbreviate $a_1, \ldots, a_k$ by $\widetilde{a}$. We use d for communication directions, α for action prefixes and P, Q for processes. Notice that message and service identifiers (from $\mathcal{L}$) are plain labels, not subject to restriction or binding. The syntax of the calculus is given in Figure 4.

The static core of our language is derived from the π-calculus [24]. We thus have $\mathbf{0}$ for the inactive process, $P \mid Q$ for the parallel composition, $(\boldsymbol{\nu} a)P$ for name restriction, and $\texttt{rec}\,\mathcal{X}.P$ and $\mathcal{X}$ for recursion. The prefix guarded choice $\Sigma_{i \in I}\, \alpha_i.P_i$ specifies a process which may exhibit any one of the α_i actions and evolve to the respective continuation P_i. Processes also specify conversation accesses: $n \blacktriangleleft [P]$ represents a process that is accessing conversation n and interacting in it according to what P specifies.

Context-oriented actions prefixes include the output $\mathtt{l}^d!(\widetilde{n})$ — send names $\widetilde{n}$ in a message labeled `l`, to either the current or enclosing conversation (depending on d); the input $\mathtt{l}^d?(\widetilde{x})$ — receive names and instantiate variables $\widetilde{x}$ in a message labeled `l` from either the current or enclosing conversation (d); and the context awareness primitive $\texttt{this}(x)$ which allows a process to dynamically gain access to the identity of its "current" ($\downarrow$) conversation.

The Conversation Calculus includes two primitives for exception handling: the `try` P `catch` Q and the $\texttt{throw}.P$. The $\texttt{throw}.P$ signals an exception and causes the termination of every process up to an enclosing `try` P `catch` Q, in which case P is activated. The `try` P `catch` Q behaves as process P up to the point an exception is thrown (if any), in which case process Q is activated.

The distinguished occurrences of a, $\widetilde{x}$, and x are binding occurrences in $(\boldsymbol{\nu} a)P$, $\mathtt{l}^d?(\widetilde{x}).P$, and $\texttt{this}(x).P$, respectively. The sets of free ($\mathit{fn}(P)$) and bound ($\mathit{bn}(P)$) names and variables in a process P are defined as usual, and we implicitly identify α-equivalent processes.

$$\begin{array}{lll}
a, b, c, \ldots \in & \Lambda & \text{(Names)} \\
x, y, z, \ldots \in & \mathcal{V} & \text{(Variables)} \\
n, v, \ldots \in & \Lambda \cup \mathcal{V} & \text{(Identifiers)} \\
\mathtt{l}, \mathtt{s} \ldots \in & \mathcal{L} & \text{(Labels)} \\
 & & \\
d & ::= \downarrow \mid \uparrow & \text{(Directions)}
\end{array}$$

$$\begin{array}{rll}
\alpha & ::= \mathtt{l}^d!(\widetilde{n}) & \text{(Output)} \\
 & \mid\ \mathtt{l}^d?(\widetilde{x}) & \text{(Input)} \\
 & \mid\ \mathtt{this}(x) & \text{(Context awareness)} \\
 & & \\
P, Q & ::= \mathbf{0} & \text{(Inaction)} \\
 & \mid\ P \mid Q & \text{(Parallel)} \\
 & \mid\ (\boldsymbol{\nu} a)P & \text{(Restriction)} \\
 & \mid\ \mathtt{rec}\, \mathcal{X}.P & \text{(Recursion)} \\
 & \mid\ \mathcal{X} & \text{(Variable)} \\
 & \mid\ \Sigma_{i \in I}\, \alpha_i.P_i & \text{(Prefix Guarded Choice)} \\
 & \mid\ n \blacktriangleleft [P] & \text{(Conversation Access)} \\
 & \mid\ \mathtt{try}\ P\ \mathtt{catch}\ Q & \text{(Try-catch)} \\
 & \mid\ \mathtt{throw}.P & \text{(Throw)}
\end{array}$$

Fig. 4. The Conversation Calculus

We define the semantics of the conversation calculus via a labeled transition system. We introduce transition labels λ and actions *act*:

$$\begin{array}{lll}
act & ::= \tau \mid \mathtt{l}^d!(\widetilde{a}) \mid \mathtt{l}^d?(\widetilde{a}) \mid \mathtt{this}^d \mid \mathtt{throw} & \text{(Actions)} \\
\lambda & ::= c\ act \mid act \mid (\nu a)\lambda & \text{(Transitions)}
\end{array}$$

Actions capture internal actions τ, message outputs $\mathtt{l}^d!(\widetilde{a})$ and inputs $\mathtt{l}^d?(\widetilde{a})$, context identity accesses $\mathtt{this}^d$ and exception signals `throw`. Transition labels tag actions with the conversation identifier they respect to $c\ act$ and with bound names which are emitted in the action $(\nu a)\lambda$. In $(\nu a)\lambda$ the distinguished occurrence of a is bound with scope λ (cf., the π-calculus bound output and bound input actions). A transition label containing $c\ act$ is said to be *located at* c (or just *located*), otherwise is said to be *unlocated.* We write $(\nu\widetilde{a})$ to abbreviate a (possibly empty) sequence $(\nu a_1)\ldots(\nu a_k)$, where the order is not important.

We adopt a few conventions and notations. We note by λ^d a transition label λ^d containing the direction d $(\uparrow, \downarrow)$. Then we denote by $\lambda^{d'}$ the label obtained by replacing d by d' in λ^d. Given an unlocated label λ, we represent by $c \cdot \lambda$ the label obtained by locating λ at c, so that e.g., $c \cdot (\nu\widetilde{a})act = (\nu\widetilde{a})c\ act$. We assert $unloc(\lambda)$ if λ is not located and is either an input or an output, and $loc(\lambda)$ if λ is located and is either an input or an output. We define $out(\lambda)$ as follows:

$$out((\nu\widetilde{a})b\ \mathtt{l}^d!(\widetilde{c})) \triangleq \widetilde{c} \setminus (\widetilde{a} \cup \{b\}) \qquad out((\nu\widetilde{a})\mathtt{l}^d!(\widetilde{c})) \triangleq \widetilde{c} \setminus (\widetilde{a})$$

and $out(\lambda) = \emptyset$ otherwise. We use $n(\lambda)$ and $bn(\lambda)$ to denote (respectively) all names and the bound names of a transition label.

The labeled transition system relies on a synchronization algebra which we now introduce. Essentially, the synchronization algebra describes how two parallel processes may synchronize, specifying how any pair of transitions may be combined and what is the result of such combination. Since not all pairs of transitions represent a valid synchronization we use the $\circ$ as the result of an invalid synchronization. We then denote by $\lambda_1 \bullet \lambda_2$ the result of combining λ_1 and λ_2 via function $\bullet$, resulting in either a transition (in case λ_1 and λ_2 may synchronize) or $\circ$ (otherwise). $\bullet$ is defined such that $\lambda_1 \bullet \lambda_2 = \lambda_2 \bullet \lambda_1$ and:

$$\begin{array}{ll}
\mathtt{l}^{\downarrow}!(\widetilde{a}) \bullet \mathtt{l}^{\downarrow}?(\widetilde{a}) & \triangleq \tau \\
\mathtt{l}^{\uparrow}!(\widetilde{a}) \bullet \mathtt{l}^{\uparrow}?(\widetilde{a}) & \triangleq \tau \\
c\,\mathtt{l}^{\downarrow}!(\widetilde{a}) \bullet c\,\mathtt{l}^{\downarrow}?(\widetilde{a}) & \triangleq \tau \\
\mathtt{l}^{\downarrow}!(\widetilde{a}) \bullet \mathtt{l}^{\uparrow}?(\widetilde{a}) & \triangleq \mathtt{this}^{\downarrow} \\
\mathtt{l}^{\downarrow}!(\widetilde{a}) \bullet c\,\mathtt{l}^{\downarrow}?(\widetilde{a}) & \triangleq c\,\mathtt{this}^{\downarrow} \\
\mathtt{l}^{\uparrow}!(\widetilde{a}) \bullet c\,\mathtt{l}^{\downarrow}?(\widetilde{a}) & \triangleq c\,\mathtt{this}^{\uparrow} \\
\mathtt{l}^{\downarrow}?(\widetilde{a}) \bullet \mathtt{l}^{\uparrow}!(\widetilde{a}) & \triangleq \mathtt{this}^{\downarrow} \\
\mathtt{l}^{\downarrow}?(\widetilde{a}) \bullet c\,\mathtt{l}^{\downarrow}!(\widetilde{a}) & \triangleq c\,\mathtt{this}^{\downarrow} \\
\mathtt{l}^{\uparrow}?(\widetilde{a}) \bullet c\,\mathtt{l}^{\downarrow}!(\widetilde{a}) & \triangleq c\,\mathtt{this}^{\uparrow}
\end{array}$$

for some $\mathtt{l}, \widetilde{a}, c$, and $\lambda_1 \bullet \lambda_2 = \circ$ otherwise. Function $\bullet$ resolves direct synchronizations (e.g., $c\,\mathtt{l}^{\downarrow}?(\widetilde{a}) \bullet c\,\mathtt{l}^{\downarrow}!(\widetilde{a})$) in an internal action τ. However, since messages are context dependent, synchronizations of unlocated transitions require contextual information. So, for instance, transition labels $\mathtt{l}^{\downarrow}!(\widetilde{a})$ and $c\,\mathtt{l}^{\downarrow}?(\widetilde{a})$ may synchronize, provided the current conversation is c — the resulting label $c\,\mathtt{this}^{\downarrow}$ will read the identity of the current conversation, and progress only if the identity is c. Intuitively, label $c\,\mathtt{this}^{\downarrow}$ captures a silent action of a process which may occur *provided* the process is placed in conversation c. Labels $\mathtt{l}^{\downarrow}?(\widetilde{a})$ and $\mathtt{l}^{\uparrow}!(\widetilde{a})$ synchronize, provided the current and enclosing conversations have the same identity — tested via label $\mathtt{this}^{\downarrow}$. Also, labels $c\,\mathtt{l}^{\downarrow}?(\widetilde{a})$ and $\mathtt{l}^{\uparrow}!(\widetilde{a})$ synchronize provided the enclosing conversation has identity c — checked via label $c\,\mathtt{this}^{\uparrow}$.

We may now present the labeled transition system. In Figs. 5, 6 and 7 we present the labeled transition system for the calculus. The rules presented in Figure 5 closely follow the π-calculus labeled transition system (see [25]). We omit the rules symmetric to (*Par*) and (*Close*).

We briefly review the rules presented in Fig. 6: in rule (*Here*), after going through a context boundary, an $\uparrow$ message becomes $\downarrow$; in (*Loc*) an unlocated $\downarrow$ message gets located at the context identity in which it originates; in (*Through*) a non-`this` located label transparently crosses the context boundary, and likewise in (*Internal*) for a τ label; in (*ThisHere*) a `this` label reads the identity of the enclosing context (and matches it with the current conversation identity); in (*ThisLoc*) a c `this` label matches the enclosing context; in (*This*) a `this` label reads the current conversation identity.

As for the rules in Figure 7: in (*Throw*) an exception is signaled; in (*ThrowPar*) and (*ThrowConv*) enclosing computations are terminated; in (*Try*) a non-`throw` transition crosses the handler block; in (*Catch*) an exception is caught by the handler block, activating (in parallel) the continuation process and the handler.

$$1^d!(\widetilde{a}).P \xrightarrow{1^d!(\widetilde{a})} P \ \ (\mathit{Out}) \qquad 1^d?(\widetilde{x}).P \xrightarrow{1^d?(\widetilde{a})} P\{\widetilde{x}\leftarrow\widetilde{a}\} \ \ (\mathit{In})$$

$$\frac{\alpha_j.P_j \xrightarrow{\lambda} Q \quad (j \in I)}{\Sigma_{i\in I}\ \alpha_i.P_i \xrightarrow{\lambda} Q}(\mathit{Sum}) \qquad \frac{P\{\mathcal{X}\leftarrow\mathtt{rec}\ \mathcal{X}.P\} \xrightarrow{\lambda} Q}{\mathtt{rec}\ \mathcal{X}.P \xrightarrow{\lambda} Q}(\mathit{Rec})$$

$$\frac{P \xrightarrow{\lambda} Q \quad (a \notin n(\lambda))}{(\boldsymbol{\nu}a)P \xrightarrow{\lambda} (\boldsymbol{\nu}a)Q}(\mathit{Res}) \qquad \frac{P \xrightarrow{\lambda} Q \quad (a \in \mathit{out}(\lambda))}{(\boldsymbol{\nu}a)P \xrightarrow{(\nu a)\lambda} Q}(\mathit{Open})$$

$$\frac{P \xrightarrow{\lambda} Q \quad (\lambda \neq \mathtt{throw})}{P \mid R \xrightarrow{\lambda} Q \mid R}(\mathit{Par}) \qquad \frac{P \xrightarrow{\lambda_1} P' \quad Q \xrightarrow{\lambda_2} Q' \quad (\lambda_1 \bullet \lambda_2 \neq \circ)}{P \mid Q \xrightarrow{\lambda_1\bullet\lambda_2} P' \mid Q'}(\mathit{Com})$$

$$\frac{P \xrightarrow{(\nu\widetilde{a})\lambda_1} P' \quad Q \xrightarrow{\lambda_2} Q' \quad (\widetilde{a} \cap \mathit{fn}(P \mid Q) = \emptyset, \lambda_1 \bullet \lambda_2 \neq \circ)}{P \mid Q \xrightarrow{\lambda_1\bullet\lambda_2} (\boldsymbol{\nu}\widetilde{a})(P' \mid Q')}(\mathit{Close})$$

Fig. 5. Transition Rules for Basic Operators

$$\frac{P \xrightarrow{\lambda^{\uparrow}} Q \quad (c \notin \mathit{bn}(\lambda))}{c \blacktriangleleft [P] \xrightarrow{\lambda^{\downarrow}} c \blacktriangleleft [Q]}(\mathit{Here}) \qquad \frac{P \xrightarrow{\lambda} Q \quad (\mathit{unloc}(\lambda))}{c \blacktriangleleft [P] \xrightarrow{c\cdot\lambda} c \blacktriangleleft [Q]}(\mathit{Loc})$$

$$\frac{P \xrightarrow{\lambda} Q \quad (\mathit{loc}(\lambda), c \notin \mathit{bn}(\lambda))}{c \blacktriangleleft [P] \xrightarrow{\lambda} c \blacktriangleleft [Q]}(\mathit{Through}) \qquad \frac{P \xrightarrow{\tau} Q}{c \blacktriangleleft [P] \xrightarrow{\tau} c \blacktriangleleft [Q]}(\mathit{Internal})$$

$$\frac{P \xrightarrow{\mathtt{this}^{\downarrow}} Q}{c \blacktriangleleft [P] \xrightarrow{c\ \mathtt{this}^{\downarrow}} c \blacktriangleleft [Q]}(\mathit{ThisHere}) \qquad \frac{P \xrightarrow{c\ \mathtt{this}^{\downarrow}} Q}{c \blacktriangleleft [P] \xrightarrow{\tau} c \blacktriangleleft [Q]}(\mathit{ThisLoc})$$

$$\mathtt{this}(x).P \xrightarrow{c\ \mathtt{this}^{\downarrow}} P\{x\leftarrow c\} \ \ (\mathit{This})$$

Fig. 6. Transition Rules for Conversation Operators

$$\mathtt{throw}.P \xrightarrow{\mathtt{throw}} P \ \ (\mathit{Throw}) \qquad \frac{P \xrightarrow{\mathtt{throw}} R}{P \mid Q \xrightarrow{\mathtt{throw}} R}(\mathit{ThrowPar})$$

$$\frac{P \xrightarrow{\mathtt{throw}} R}{n \blacktriangleleft [P] \xrightarrow{\mathtt{throw}} R}(\mathit{ThrowConv}) \qquad \frac{P \xrightarrow{\mathtt{throw}} R}{\mathtt{try}\ P\ \mathtt{catch}\ Q \xrightarrow{\tau} Q \mid R}(\mathit{Catch})$$

$$\frac{P \xrightarrow{\lambda} Q \qquad \lambda \neq \mathtt{throw}}{\mathtt{try}\ P\ \mathtt{catch}\ R \xrightarrow{\lambda} \mathtt{try}\ Q\ \mathtt{catch}\ R}(\mathit{Try})$$

Fig. 7. Transition Rules for Exception Handling Operators

$$\begin{array}{ll}
\texttt{def}\, s \Rightarrow P & \triangleq \mathtt{s}^{\downarrow}?(x).x \blacktriangleleft [P] \\
\texttt{new}\, n \cdot s \Leftarrow Q & \triangleq (\boldsymbol{\nu} c)(n \blacktriangleleft \left[\mathtt{s}^{\downarrow}!(c)\right] \mid c \blacktriangleleft [Q]) \\
\texttt{join}\, n \cdot s \Leftarrow Q & \triangleq \mathtt{this}(x).(n \blacktriangleleft \left[\mathtt{s}^{\downarrow}!(x)\right] \mid Q) \\
\star\texttt{def}\, s \Rightarrow P & \triangleq \mathtt{rec}\, \mathcal{X}.\mathtt{s}^{\downarrow}?(x).(\mathcal{X} \mid x \blacktriangleleft [P])
\end{array}$$

Fig. 8. Service Idioms

Notice that the presentation of the transition system is fully modular: the rules for each operator are independent, so that one may easily consider several fragments of the calculus (e.g., without exception handling primitives). The operational semantics of closed systems, usually represented by a reduction relation, is here specified by $\xrightarrow{\tau}$.

3.5 Representing Service-Oriented Idioms

Our core model focuses on the fundamental notions of conversation context and message-based communication. From these basic mechanisms, useful programming abstractions for service-oriented systems may be idiomatically defined, namely service definition and instantiation constructs (defined as primitives in [26]), and the conversation join construct (introduced in [13]). These constructs may be embedded in a simple way in the minimal calculus, without hindering the flexibility of modeling and analysis.

The service-oriented idioms along with their translation in lower level communication primitives is shown in Fig. 8. We specify the service definition idiom by $\texttt{def}\, s \Rightarrow P$, which publishes a service named s in the current conversation. Process P specifies the code that will run in the service conversation, upon service instantiation, implementing the service provider role in the conversation. The service definition is implemented in basic communication primitives as:

$$\texttt{def}\, s \Rightarrow P \;\triangleq\; \mathtt{s}^{\downarrow}?(x).x \blacktriangleleft [P] \qquad (x \notin \mathit{fv}(P))$$

Essentially, the service definition specifies a message — labeled by the name of the service s — is received, carrying the identity of the service conversation. Then, code P will run in such received conversation. Service definitions must be placed in appropriate conversation contexts (cf., methods in objects). For instance, to specify `BuyService` is published in the *Seller* context we write:

$$\mathit{Seller} \blacktriangleleft [\,\texttt{def BuyService} \Rightarrow \mathit{SellerCode}\,]$$

Typically, services once published are persistent in the sense they can be instantiated several times. To model such persistent services we introduce the recursive variant of service definition.

$$\star\texttt{def}\, s \Rightarrow P \;\triangleq\; \mathtt{rec}\, \mathcal{X}.\mathtt{s}^{\downarrow}?(x).(\mathcal{X} \mid x \blacktriangleleft [P]) \qquad (x \notin \mathit{fv}(P))$$

Persistent service definitions are specified so as to always be ready to receive a service instantiation request, handling each request in the conversation received in each service instantiation message.

The idiom that supports the instantiation of a published service is noted by $\texttt{new}\ n \cdot s \Leftarrow Q$. The `new` idiom specifies the conversation where the service is published at (n), the name of the service (s) and the code that will run on the service client side (Q). A service instantiation resulting from a synchronization from a published service `def` and an instantiation `new` results in the creation of a fresh conversation that is shared between service provider and service caller. We translate the `new` idiom in the basic primitives of the CC as follows:

$$\texttt{new}\ n \cdot s \Leftarrow Q \ \triangleq\ (\boldsymbol{\nu} c)(n \blacktriangleleft \left[\texttt{s}^{\downarrow}!(c)\right] \mid c \blacktriangleleft [Q]) \qquad (c \notin (fn(Q) \cup \{n\}))$$

The service instantiation is then realized by means of a message exchange in conversation n, where the service is published at, being the message labeled by the name of the service s and carrying a newly created name c that identifies the conversation where the service interaction is to take place. In parallel to the message output that instantiates the service, we find the code of the client role Q, running in the freshly created conversation c. Notice that Q is already active, although it has to wait for the server side to pick up the service conversation identity to start interacting in conversation c, by means of $\downarrow$ directed messages. Notice also that process Q can interact in the conversation where the service instantiation request lies, using $\uparrow$ directed messages.

The join idiom is implemented using the core CC primitives as follows:

$$\texttt{join}\ n \cdot s \Leftarrow Q \ \triangleq\ \texttt{this}(x).(n \blacktriangleleft \left[\texttt{s}^{\downarrow}!(x)\right] \mid Q) \qquad (x \notin (fv(Q) \cup \{n\}))$$

The current conversation identity is accessed via the `this` primitive, and passed along in service message s exchanged in the conversation n where s is published at. Process Q continues to interact in the current conversation (the same that was accessed in the `this`).

4 A Sequence of Examples

In this section, we illustrate the expressiveness of our calculus through a sequence of examples. For the sake of commodity, we informally extend the language with some auxiliary primitives, e.g., `if` − `then` − `else`, etc. We also use replication !, which may be simulated using recursion, anonymous contexts, defined as $[P] \triangleq (\boldsymbol{\nu} a)(a \blacktriangleleft [P])$ (where a is fresh) to isolate communication, and we omit $\downarrow$ message directions (e.g., $\texttt{read}?()$ abbreviates $\texttt{read}^{\downarrow}?()$).

4.1 Memory Cell

We discuss some simple examples of stateful service definition and invocation patterns, using memory cell implementations. Consider the following implementation of a memory cell service.

```
def Cell ⇒ (
     !(read?().next!()
       +
      write?(x).stop!().rec 𝒳.(stop?() + next?().value!(x).𝒳)
      |
     stop?())
```

Intuitively, each time a value is written in the cell, a process that stores the value is spawned. This process is ready to repeatedly emit the value upon request (message next), or to stop giving out the value. To read the cell value, a request for the next emission of the value is sent to the memory process. To write a new value, the installed memory process is stopped, and another process which stores the new value is spawned (notice that since the first write does not have to stop any memory process, the respective stop message is collected separately).

We show how to instantiate the Cell service so to create a delegate cell process in the current context. The delegate accepts put and get messages from the client context, and replies to each get message with a reply message to the context. It provides the memory cell functionality delegation to the remote service *FreeCellsInc* ◀ Cell.

```
new FreeCellsInc · Cell ⇐ (
     !(put↑?(x).write!(x)
       +
      get↑?().read!().value?(x).reply↑!(x)) )
```

A process in the context may then use the created service instance as follows:

```
put!(value).get!().reply?(x).proceed!(x)
```

To show how the system evolves, let us consider the composition of the Cell service provider and user, placing the provider in the *FreeCellsInc* context:

```
FreeCellsInc ◀ [
     def Cell ⇒ ( !(read?().next!()
                     +
                    write?(x).stop!().rec 𝒳.(stop?() + next?().value!(x).𝒳)
                    |
                   stop?() ]
 |
new FreeCellsInc · Cell ⇐ (!(put↑?(x).write!(x)
                               +
                              get↑?().read!().value?(x).reply↑!(x)))
 |
put!(value).get!().reply?(x).proceed!(x)
```

By translating the service idioms into their lower level representation we obtain:

$$\begin{array}{l}
\mathit{FreeCellsInc} \blacktriangleleft [\\
\quad \mathtt{Cell?}(y).y \blacktriangleleft [\ !(\mathtt{read?}().\mathtt{next!}()\\
\qquad\qquad\qquad +\\
\qquad\qquad\qquad \mathtt{write?}(x).\mathtt{stop!}().\mathtt{rec}\,\mathcal{X}.(\mathtt{stop?}() + \mathtt{next?}().\mathtt{value!}(x).\mathcal{X})\\
\qquad\qquad\qquad |\\
\qquad\qquad\quad \mathtt{stop?}()\ ]\ ]\\
|\\
(\boldsymbol{\nu} c)(\mathit{FreeCellsInc} \blacktriangleleft [\mathtt{Cell!}(c)]\\
\quad |\\
\quad c \blacktriangleleft [\ !(\mathtt{put}^{\uparrow}\mathtt{?}(x).\mathtt{write!}(x)\\
\qquad\quad +\\
\qquad\quad \mathtt{get}^{\uparrow}\mathtt{?}().\mathtt{read!}().\mathtt{value?}(x).\mathtt{reply}^{\uparrow}\mathtt{!}(x))\ ]\ )\\
|\\
\mathtt{put!}(\mathit{value}).\mathtt{get!}().\mathtt{reply?}(x).\mathtt{proceed!}(x)
\end{array}$$

Service provider and client may synchronize in the `Cell` service message, being (fresh) name c passed in the message which allows the service provider to gain access to the conversation.

$$\begin{array}{l}
(\boldsymbol{\nu} c)(\mathit{FreeCellsInc} \blacktriangleleft [\\
\quad c \blacktriangleleft [\ !(\mathtt{read?}().\mathtt{next!}()\\
\qquad\quad +\\
\qquad\quad \mathtt{write?}(x).\mathtt{stop!}().\mathtt{rec}\,\mathcal{X}.(\mathtt{stop?}() + \mathtt{next?}().\mathtt{value!}(x).\mathcal{X})\\
\qquad\quad |\\
\qquad \mathtt{stop?}()\ ]\ ]\\
|\\
c \blacktriangleleft [\ !(\mathtt{put}^{\uparrow}\mathtt{?}(x).\mathtt{write!}(x)\\
\qquad +\\
\qquad \mathtt{get}^{\uparrow}\mathtt{?}().\mathtt{read!}().\mathtt{value?}(x).\mathtt{reply}^{\uparrow}\mathtt{!}(x))\ ]\ )\\
|\\
\mathtt{put!}(\mathit{value}).\mathtt{get!}().\mathtt{reply?}(x).\mathtt{proceed!}(x)
\end{array}$$

The service client instance and the process using it interact in message `put` (notice the $\uparrow$ direction), activating a cell write.

$$\begin{array}{l}
(\boldsymbol{\nu} c)(\mathit{FreeCellsInc} \blacktriangleleft [\\
\quad c \blacktriangleleft [\ !(\mathtt{read?}().\mathtt{next!}()\\
\qquad\quad +\\
\qquad\quad \mathtt{write?}(x).\mathtt{stop!}().\mathtt{rec}\,\mathcal{X}.(\mathtt{stop?}() + \mathtt{next?}().\mathtt{value!}(x).\mathcal{X})\\
\qquad\quad |\\
\qquad \mathtt{stop?}()\ ]\ ]\\
|\\
c \blacktriangleleft [\ !(\mathtt{put}^{\uparrow}\mathtt{?}(x).\mathtt{write!}(x) + \mathtt{get}^{\uparrow}\mathtt{?}().\mathtt{read!}().\mathtt{value?}(x).\mathtt{reply}^{\uparrow}\mathtt{!}(x))\\
\qquad |\\
\qquad \mathtt{write!}(\mathit{value})\ ]\ )\\
|\\
\mathtt{get!}().\mathtt{reply?}(x).\mathtt{proceed!}(x)
\end{array}$$

At this point, service provider and client instances exchange message write in the service conversation c, after which message stop is exchanged.

$$
\begin{array}{l}
(\boldsymbol{\nu} c)(\mathit{FreeCellsInc} \blacktriangleleft [\\
\quad c \blacktriangleleft [\ !(\mathtt{read}?().\mathtt{next}!()\\
\qquad\quad +\\
\qquad\quad \mathtt{write}?(x).\mathtt{stop}!().\mathtt{rec}\,\mathcal{X}.(\mathtt{stop}?() + \mathtt{next}?().\mathtt{value}!(x).\mathcal{X})\\
\qquad\quad |\\
\qquad\quad \mathtt{rec}\,\mathcal{X}.(\mathtt{stop}?() + \mathtt{next}?().\mathtt{value}!(\mathit{value}).\mathcal{X})\ ]\ ]\\
\ |\\
c \blacktriangleleft [\ !(\mathtt{put}^{\uparrow}?(x).\mathtt{write}!(x)\\
\qquad +\\
\qquad \mathtt{get}^{\uparrow}?().\mathtt{read}!().\mathtt{value}?(x).\mathtt{reply}^{\uparrow}!(x))\ ]\ )\\
\ |\\
\mathtt{get}!().\mathtt{reply}?(x).\mathtt{proceed}!(x)
\end{array}
$$

Then, the `get` message is exchanged between service client instance and its user process, activating a cell read.

$$
\begin{array}{l}
(\boldsymbol{\nu} c)(\mathit{FreeCellsInc} \blacktriangleleft [\\
\quad c \blacktriangleleft [\ !(\mathtt{read}?().\mathtt{next}!()\\
\qquad\quad +\\
\qquad\quad \mathtt{write}?(x).\mathtt{stop}!().\mathtt{rec}\,\mathcal{X}.(\mathtt{stop}?() + \mathtt{next}?().\mathtt{value}!(x).\mathcal{X})\\
\qquad\quad |\\
\qquad\quad \mathtt{rec}\,\mathcal{X}.(\mathtt{stop}?() + \mathtt{next}?().\mathtt{value}!(\mathit{value}).\mathcal{X})\ ]\ ]\\
\ |\\
c \blacktriangleleft [!(\mathtt{put}^{\uparrow}?(x).\mathtt{write}!(x) + \mathtt{get}^{\uparrow}?().\mathtt{read}!().\mathtt{value}?(x).\mathtt{reply}^{\uparrow}!(x))\\
\qquad |\\
\qquad \mathtt{read}!().\mathtt{value}?(x).\mathtt{reply}^{\uparrow}!(x))\ ]\ )\\
\ |\\
\mathtt{reply}?(x).\mathtt{proceed}!(x)
\end{array}
$$

At this point, service provider and client exchange message `read`, after which message `next` is exchanged and the value emission is activated.

$$
\begin{array}{l}
(\boldsymbol{\nu} c)(\mathit{FreeCellsInc} \blacktriangleleft [\\
\quad c \blacktriangleleft [\ !(\mathtt{read}?().\mathtt{next}!()\\
\qquad\quad +\\
\qquad\quad \mathtt{write}?(x).\mathtt{stop}!().\mathtt{rec}\,\mathcal{X}.(\mathtt{stop}?() + \mathtt{next}?().\mathtt{value}!(x).\mathcal{X})\\
\qquad\quad |\\
\qquad\quad \mathtt{value}!(\mathit{value}).\mathtt{rec}\,\mathcal{X}.(\mathtt{stop}?() + \mathtt{next}?().\mathtt{value}!(\mathit{value}).\mathcal{X})\ ]\ ]\\
\ |\\
c \blacktriangleleft [!(\mathtt{put}^{\uparrow}?(x).\mathtt{write}!(x)\\
\qquad +\\
\qquad \mathtt{get}^{\uparrow}?().\mathtt{read}!().\mathtt{value}?(x).\mathtt{reply}^{\uparrow}!(x))\\
\qquad |\\
\qquad \mathtt{value}?(x).\mathtt{reply}^{\uparrow}!(x))\ ]\ )\\
\ |\\
\mathtt{reply}?(x).\mathtt{proceed}!(x)
\end{array}
$$

Now, the `value` message is exchanged, after which message `reply` carrying the initially written value is picked up by the user process allowing it to `proceed`.

4.2 Dictionary

In the next example, we use a toy dictionary service to discuss the possible need of correlating messages belonging to different interaction contexts. A possible instantiation of such a service may be expressed thus:

$$\begin{array}{l} \texttt{new}\ \mathit{FreeBagsCo} \cdot \texttt{Dict} \Leftarrow (\\ \quad !(\texttt{put}^{\uparrow}?(\mathit{key}, x).\texttt{store}!(\mathit{key}, x) \\ \quad\ + \\ \quad\ \texttt{get}^{\uparrow}?(\mathit{key}).\texttt{get}!(\mathit{key}).\texttt{value}?(x).\texttt{reply}^{\uparrow}!(x) \\ ) \end{array}$$

If the generated instance is to be solicited by several concurrent `get` requests, some form of correlation may be needed, in order to route the associated `reply` answers to the appropriate contexts. In this case, we set the `get` message to play the role of an initiator message, now receiving also a reference r to the context of interaction (associated to getting the dictionary entry associated to the indicated key).

$$\begin{array}{l} \texttt{new}\ \mathit{FreeBagsCo} \cdot \texttt{Dict} \Leftarrow (\\ \quad !(\texttt{put}^{\uparrow}?(\mathit{key}, x).\texttt{store}!(\mathit{key}, x) \\ \quad\ + \\ \quad\ \texttt{get}^{\uparrow}?(r, \mathit{key}).\texttt{get}!(\mathit{key}).\texttt{value}?(x).r \blacktriangleleft [\texttt{reply}!(x)] \\ ) \end{array}$$

Now, the `reply` message is sent inside the appropriate conversation context r, the one that relates to the initial `get`. A process in the context may then use the service instance by following the appropriate intended protocol, e.g.:

$$\texttt{put}!(\mathit{key}, \mathit{value}).(\boldsymbol{\nu} r)(\texttt{get}(r, \mathit{key}).r \blacktriangleleft [\texttt{reply}?(x).\texttt{proceed}^{\uparrow}!(x)])$$

Here, we are essentially in presence of a familiar form of continuation passing.

In this case, we have generated a new special context r in order to carry out the appropriate conversation. In many situations we would like just to correlate the subsidiary conversation with the current context, without having to introduce a new special context. In this case, we may write the (perhaps more natural) code, that will have the same effect as the code above:

$$\texttt{put}!(\mathit{key}, \mathit{value}).\texttt{this}(\mathit{currentC}).\texttt{get}(\mathit{currentC}, \mathit{key}).\texttt{reply}?(x).\texttt{proceed}!(x)$$

Remember that the $\texttt{this}(x).P$ (context-awareness) primitive binds x in P to the identity of the current context.

4.3 Service Provider Factory

We revisit the memory cell example, and provide a different realization. In this case, we would like to represent each cell as a specific service provider, such that the `Read` and `Write` operations are now services, rather than operations of a particular service as shown above. A cell (named n) may be be represented by the context:

$$\begin{aligned} \mathit{Cell}(n) \triangleq n \blacktriangleleft [\ & \texttt{def Read} \Rightarrow \texttt{value}^{\uparrow}?(x).\texttt{value}!(x) \mid \\ & \texttt{def Write} \Rightarrow \texttt{value}?(x).\texttt{value}^{\uparrow}!(x)\ \ ] \end{aligned}$$

We may now specify a memory cell factory service.

$$\mathit{CellFactoryService} \triangleq \texttt{def NewCell} \Rightarrow (\boldsymbol{\nu} a)(\mathit{Cell}(a) \mid \texttt{replyCell}!(a))$$

To instantiate the cell factory service, and drop a `theCell` message with a fresh cell reference (c) in the current context, we may write:

$$\texttt{new}\ \mathit{FreeCellsInc} \cdot \texttt{NewCell} \Leftarrow \texttt{replyCell}?(x).\texttt{theCell}^{\uparrow}!(x)$$

The newly allocated cell service provider is allocated in the *FreeCellsInc* context, as expected. To use the cell one may then write, e.g.,

$$\begin{array}{l} \texttt{theCell}(c).(\\ \cdots \\ \texttt{new}\ c \cdot \mathit{Read} \Leftarrow \cdots \\ \ \mid \cdots \\ \texttt{new}\ c \cdot \mathit{Write} \Leftarrow \cdots \\ \cdots) \end{array}$$

This usage pattern for services, where service instantiation corresponds to some form of task delegation rather that process delegation, is closer to a distributed object model than to a service-oriented model. In any case, it is interesting to be able to accommodate this usage pattern as a special case, not only for the sake of abstract generality, but also because it will certainly turn out useful in appropriate scenarios.

4.4 Exceptions

We illustrate a few usage idioms for our exception handling primitives. In the first example, the service `Service` is instantiated on site *Server*, and repeatedly re-launched on each failure – failure will be signaled by exception throwing within the local protocol *ClientProto*, possibly as a result of a remote message.

$$\begin{array}{l} \texttt{rec}\ \mathit{Restart}. \\ \quad \texttt{try} \\ \quad\quad \texttt{new}\ \mathit{Server} \cdot \texttt{Service} \Leftarrow \mathit{ClientProto} \\ \quad \texttt{catch}\ \mathit{Restart} \end{array}$$

A possible scenario of remote exception throwing is illustrated below.

```
Server ◀ [
   def Interruptible ⇒
     stop?().urgentStop!().throw |
     ... ServiceProto ...]

new Server · Interruptible ⇐
   urgentStop?().throw |
   ... ClientProto ...
```

Here, any remote endpoint instance of the `Interruptible` service may be interrupted by the service protocol *ServiceProto* by dropping a message `stop` inside the endpoint context. In this example, such a message causes the endpoint to send an `urgentStop` message to the client side, and then throwing an exception, which will cause abortion of the service endpoint. On the other hand, the service invocation protocol will throw an exception at the client endpoint upon reception of `urgentStop`. Notice that this behavior will happen concurrently with ongoing interactions between *ServiceProto* and *ClientProto*. In this example, the exception issued in the server and client endpoints will have to be managed by appropriate handlers in both sites. In the next example, no exception will be propagated to the service site, but only to the remote client endpoint, and as a result of any exception thrown in the service protocol *ServiceProto*.

```
Server ◀ [
   def Interruptible ⇒
     try
       ServiceProto
     catch urgentStop!().throw ]
```

In the examples discussed above, the decision to terminate the ongoing remote interactions is triggered by the service code. In the next example, we show a simple variation of the idioms above, where the decision to kill the ongoing service instance is responsibility of the service context. Here, any instance of the `Interruptible` service may be terminated by the service provider by means of dropping a message `killRequest` in the endpoint external context.

```
Server ◀ [
   def Interruptible ⇒
     try
       killRequest↑?().throw | ServiceProto
     catch urgentStop().throw ]
```

A simple example of a similar pattern in our last example on exceptions.

```
Server ◀ [
   def TimeBound ⇒
       timeAllowed↑?(delay).wait(delay).throw |
       ServiceProto ]
```

Here, any invocation of the `TimeBound` service will be allocated no more than *delay* time units before being interrupted, where *delay* is a dynamic parameter value read from the current server side context (we assume some extension of our sample language with a `wait`(t) primitive, with the expected semantics).

4.5 Programming a Finance Portal

In this section we show the implementation of a Finance portal (inspired in a Sensoria Project [21] case study) which illustrates how the several primitives and idioms of the language can be combined, allowing to model complex interaction patterns in a rather simple way. We model a credit request scenario, where a bank client, a bank clerk and a bank manager participate, mediated through a bank portal. The client starts by invoking a service available in the bank portal and places the credit request, providing his identification and the desired amount. The implementation of such client in CC is then:

```
Client ◀ [ ClientTerminal
    |
  new BankPortal · CreditRequest ⇐
     request!(myId, amount).
        (requestApproved?().transferDate!(date).approved↑!()
        +
        requestDenied?().denied↑!())  ]
```

The client code for the service instantiation specifies the messages that are to be exchanged in the service conversation by using ↓ messages. First a message `request` is sent, after which one of two messages (either `requestApproved` or `requestDenied`) informing on the decision is received. Only after receiving one of such messages is the *ClientTerminal* informed (correspondingly) of the final decision. Notice that the service code interacts with the *ClientTerminal* process by means of ↑ messages `approved` or `denied`. In fact, from the point of view of *ClientTerminal* the external interface of the service instance can be characterized by the process `approved!()` + `denied!()`.

Next we show the code of the `CreditRequest` service published in conversation *BankPortal*, and persistently available (as indicated by the ⋆ annotation).

```
BankPortal ◀ [⋆ def CreditRequest ⇒
                 request?(uid, amount).
                 join Clerk · RiskAssessment ⇐
                    assessRisk!(uid, amount).
                    riskVal?(risk).
                    if risk = HIGH then requestDenied!()
                    else this(clientC).
                       new Manager · CreditApproval ⇐
                          requestApproval!(clientC, uid, amount, risk)  ]
```

The server code specifies that, in each `CreditRequest` service conversation, a message `request` is received, then message `assessRisk` is sent and then message `riskVal` is received. The first will be exchanged with the service client, while the latter two will be exchanged with the clerk, that is asked to join the ongoing conversation through service `RiskAssessment`. After that, depending on the risk rate the clerk determined for the request, the bank portal is either able to automatically reject the request, in which case it informs the client of such decision by sending message `requestDenied`, or it has to consult the bank manager, creating a new instance of the `CreditApproval` service to that end — notice that a `new` instance is created in this case. However, since the bank manager will reply directly back to the client, the name of the client service conversation is accessed, via the `this`(*clientC*), and passed along to the manager (in the first argument of message `requestApproval`). This pattern is similar to a `join`: the name of the current conversation is sent to the remote service provider, allowing for it to join in the conversation. The difference with respect to a join is that the remote service will only join the client conversation to reply back to the client. In some sense, it is as if we only delegate a basic fragment of the client conversation (e.g., the final reply), instead of incorporating the whole functionality provided by `CreditApproval` in the `CreditRequest` service collaboration.

We now show the code for the `CreditApproval` service, assuming there is a *ManagerTerminal* process able to interact with the manager, similarly to the *ClientTerminal* process.

```
Manager ◀ [ ManagerTerminal
    |
    ⋆ def CreditApproval ⇒
        requestApproval?(clientC, uid, amount, risk).
        this(managerC).
        showRequest↑!(managerC, uid, amount, risk).
            (reject?().clientC ◀ [ requestDenied!() ]
            +
            accept?().clientC ◀ [ requestApproved!().
                                    join BankATM · CreditTransfer ⇐
                                        orderTransfer!(uid, amount) ] ]
```

The `CreditApproval` server code specifies the reception of a `requestApproval` message, carrying the name of the conversation where the final answer is to be given in, after which the identity of the current conversation is accessed, and passed along to *ManagerTerminal* in message `showRequest` in conversation *Manager*. This allows *ManagerTerminal* to reply directly to the "right" conversation, since several copies of the `CreditApproval` service may be running in parallel, and therefore several `showRequest` messages may have to be concurrently handled and replied to by the `ManagerTerminal`: if the replies were to be placed in the *Manager* conversation then they would also compete and be at risk of being picked up by the wrong (unrelated) service instance. The *ManagerTerminal* thus replies in the `CreditApproval` service conversation with either a `reject` message or an `accept` message. After that the credit request client is notified accordingly in the respective conversation. Also, in the case that the credit is approved, the manager asks service `CreditTransfer` published at

BankATM to join the client conversation (the current conversation for the `join` is the client conversation), so as to place the transfer order.

We now specify the code for the *CreditTransfer* service.

$$
\begin{array}{l}
BankATM \blacktriangleleft [\\
\quad BankATMProcess\\
\quad |\\
\quad \star\ \texttt{def CreditTransfer} \Rightarrow\\
\qquad \texttt{orderTransfer}?(uid, amount).\\
\qquad \texttt{transferDate}?(date).\\
\qquad \texttt{scheduleTransfer}^{\uparrow}!(uid, amount, date)\ \ ]
\end{array}
$$

The `CreditTransfer` service code specifies the reception of the transfer order and of the desired date of the transfer, after which forwards the information to a local *BankATMProcess*, which will then schedule the necessary procedure. Notice that the *BankATM* party is only asked to join in the conversation under some circumstances, in such case interacting with the bank manager in message `orderTransfer` and with the credit request client in message `transferDate`, while otherwise it does not participate at all in the service collaboration.

The system obtained by composing the described processes captures an interesting scenario where, not only the set of multiple participants in the collaboration is dynamically determined, but also the actual maximum number of participants depends on a runtime condition.

5 Analysis Techniques

In several works, we have studied the dynamic and static semantics of the CC, and illustrated their use to the analysis of service-based systems. Namely, we have investigated behavioral equivalences, and type systems for conversation fidelity and deadlock absence. In this tutorial note, we focus on basic results of the observational semantics; in Section 6, we give further pointers to the type based analysis.

We define a compositional behavioral semantics of the conversation calculus by means of the standard notion of strong bisimulation. We prove that strong and weak bisimilarity are congruences for all the primitives of our calculus. This further ensures that our syntactically defined constructions induce properly defined behavioral operators at the semantic level.

Definition 1. *A (strong) bisimulation is a symmetric binary relation $\mathcal{R}$ on processes such that, for all processes P and Q, if $P\mathcal{R}Q$, we have:*

If $P \xrightarrow{\lambda} P'$ and $bn(\lambda) \cap fn(P \mid Q) = \emptyset$ then there is a process Q' such that $Q \xrightarrow{\lambda} Q'$ and $P'\mathcal{R}Q'$.

We denote by $\sim$ (strong bisimilarity) the largest strong bisimulation.

Strong bisimilarity is an equivalence relation. We also have:

Theorem 1. *Strong bisimilarity is a congruence.*

We consider weak bisimilarity defined as usual, denoted by $\approx$.

Theorem 2. *Weak bisimilarity is a congruence.*

Notice Theorem 2 is not a direct consequence of Theorem 1. In fact, there are other languages where the latter holds while the former does not. Informally, the usual counter-example is given by processes $\tau.\alpha.P$ and $\alpha.P$ which are weakly bisimilar. Put in a summation context with process R we obtain $\tau.\alpha.P + R$ and $\alpha.P + R$ which are not weakly bisimilar (the former can do a silent action and lose the ability to do R and the latter cannot mimic such action).

We also prove other interesting behavioral equations, for instance, the following equations hold up to strong bisimilarity:

1. $n \blacktriangleleft [P] \mid n \blacktriangleleft [Q] \sim n \blacktriangleleft [P \mid Q]$.
2. $m \blacktriangleleft [n \blacktriangleleft [o \blacktriangleleft [P]]] \sim n \blacktriangleleft [o \blacktriangleleft [P]]$.

(1) captures the local character of message-based communication in our model, while (2) illustrates the idea that processes located in different access pieces of the same conversation interact as if they where located in the same access piece. Using such behavioral identities, in [26] we show that processes admit a flat representation, where the nesting level of any active communication prefix is at most two.

As an example of the sort of specifications captured by our type analysis consider the `CreditApproval` service shown in Section 4.5. After being notified by of the credit approval decision, the `CreditApproval` service instance forwards the notification to the client and, in case of approval, asks service `CreditTransfer` to join the ongoing conversation. The type that captures the role of the `CreditApproval` service instance in the client conversation is characterized by the following type (assuming some basic types $\mathtt{T}_1, \ldots$):

$$\begin{aligned} &\mathit{managerR} \triangleq \\ &\quad \oplus\{!\,\mathtt{requestApproved}().\boldsymbol{\tau}\,\mathtt{orderTransfer}(\mathtt{T}_1,\mathtt{T}_2).?\,\mathtt{transferDate}(\mathtt{T}_4);\\ &\qquad !\,\mathtt{requestDenied}()\}\end{aligned}$$

Type *managerR* specifies a choice ($\oplus$) between two outputs (!): either the output of message `requestApproved` or of message `requestDenied`. In the case of the `requestApproved` choice, the type specifies that the process proceeds by internally exchanging ($\boldsymbol{\tau}$) message `orderTransfer` and by receiving message `transferDate`. This behavior results from the combination of the rest of the manager role in the client conversation (the output of message `orderTransfer`, typed $!\,\mathtt{orderTransfer}(\mathtt{T}_1,\mathtt{T}_2)$) with the type of the `CreditTransfer` service:

$$\mathit{creditTransferB} \triangleq ?\,\mathtt{orderTransfer}(\mathtt{T}_1,\mathtt{T}_2).?\,\mathtt{transferDate}(\mathtt{T}_4)$$

where the synchronization in message `orderTransfer` between the manager and the bank is recorded in *managerR*, using the $\boldsymbol{\tau}$ annotation. Such flexibility in combining behavioral types is crucial to capture conversations between several parties, including scenarios where parties dynamically join and leave conversations, which is the case of the `CreditTransfer` service.

The `CreditApproval` service is then characterized by the type:

$$\begin{aligned}&\mathit{creditApprovalB} \triangleq \\ &\quad ?\,\mathtt{requestApproval}(\mathit{managerR}, \mathtt{T}_1, \mathtt{T}_2, \mathtt{T}_3).\oplus\{\boldsymbol{\tau}\,\mathtt{reject}();\ \boldsymbol{\tau}\,\mathtt{accept}()\}\end{aligned}$$

which specifies the reception of a `requestApproval` message, carrying a conversation identifier in which the manager behaves as specified by *managerR*, after which proceeding as the internal choice between messages `reject` and `accept`.

The type of the manager process exposes the required and provided services:

$$\begin{aligned}&\mathit{ManagerProcess} :: \\ &\quad \mathit{Manager} : [\star ?\,\mathtt{CreditApproval}(\mathit{creditApprovalB}) \\ &\quad \mid \\ &\quad \mathit{BankATM} : [\star !\,\mathtt{CreditTransfer}(\mathit{creditTransferB})]\end{aligned}$$

service `CreditApproval` is published (?) in conversation *Manager*, and service `CreditTransfer` is expected (!) in conversation *BankATM*.

6 Further Reading and Closing Remarks

The Conversation Calculus was first introduced in [26], where we also presented a basic study of its behavioral semantics. In [13,14] we introduced the conversation type theory, which provides analysis techniques for conversation fidelity and deadlock absence, while addressing challenging scenarios involving dynamically established conversations between several partners. Analysis techniques based on the CC where mainly developed by Hugo Vieira in his PhD thesis [27]. Several aspects of the Conversation Calculus have also been reported in several chapters of the book which collects the key results of IP SENSORIA Project [1,3,11,17,22]. More recently, in the context of the CMU-PT INTERFACES project [15], we have been using the Conversation Calculus/Types suite to model and analyze role based multiparty interactions in the setting of web business applications.

Our development of the concept of conversation was initially motivated by the concept of binary session [19]; most session-based approaches to service interactions only support binary interactions (simple client-server). Only recently proposals have appeared to support multiparty interaction [5,6,16,20,28]. To support multiparty interaction, [20] considers multiple session channels, while [5] considers indexed session channels, both resorting to multiple communication pathways. Our model follows an essentially different design, by letting a single medium of interaction support concurrent multiparty interaction via labeled messages. We base our approach on the notion of conversation context, and on plain message-passing communication, which allows us to introduce the conversation initiation and conversation join constructs as idioms. This contrasts with other session-based proposals for session and service-oriented models where we find primitive service instantiation operations constructs (see, e.g., [7,8,23]). Comparing with such models, the Conversation Calculus seems to be the simplest extension to the pure π-calculus for modeling and analyzing complex multi-party service based systems.

Ad hoc primitives to deal with exceptional behavior are present in several service calculi. Perhaps surprisingly, our exception mechanism, although clearly based on the canonical construct for functional languages, does not seem to have been explored before us in process calculi (more recently, a similar mechanism is explored in [9]). In [12] we showed how a transactional model supporting compensations (the compensating CSP [10]) can be encoded in the Conversation Calculus by means of its exception mechanism.

In our discussion on the underlying principles of the service-oriented computational model, we left out some interesting features of distributed systems that we view as fairly alien to this setting. Forms of code migration (weak mobility) seem to require an homogeneous execution support infrastructure, and thus to run against the aim to get loose coupling, openness and independent evolution of subsystems. In general, any mechanism relying on centralized control or authority mechanisms, or that require a substantial degree on homogeneity in the runtime infrastructure (e.g., strong mobility) also seem hard to accommodate.

To summarize, we have reviewed the Conversation Calculus, a core model for service-oriented computation. The design of the Conversation Calculus, building on the identification and analysis of general aspects of service-based systems, was also discussed and justified in considerable detail. By means of a series of simple, yet hopefully illuminating examples, we have illustrated how our model may express many service-oriented idioms, and complex multi-party service based systems in a very natural way. Properties such as behavioral equivalence, conversation fidelity and deadlock absence may be verified on Conversation Calculus models by means of several available techniques. The aim of providing simpler, more expressive, and usable techniques for complex software systems is certainly a good justification for introducing yet another core model or language, in particular if such model is expressed as a tiny layer on top of a purest foundational model; the π calculus. We leave for others to judge the extent to which such aim was achieved by the Conversation Calculus and related techniques.

Acknowledgments. We thank IP Sensoria EU IST FP6 - 2005-2010 and Carnegie Mellon|PT INTERFACES 44-2009-12. We also thank Luís Barbosa and Markus Lumpe for inviting us for the FACS'10 keynote, on which this tutorial is based.

References

1. Acciai, L., Bodei, C., Boreale, M., Bruni, R., Vieira, H.: Static analysis techniques for session-oriented calculi. In: Hölzl, M. (ed.) SENSORIA Project. LNCS, vol. 6582, pp. 214–231. Springer, Heidelberg (2011)
2. Alves, A., et al.: Web Services Business Process Execution Language Version 2.0. Technical report, OASIS (2006)

3. Bartoletti, M., Caires, L., Lanese, I., Mazzanti, F., Sangiorgi, D., Vieira, H., Zunino, R.: Tools and verification. In: Hölzl, M. (ed.) SENSORIA Project. LNCS, vol. 6582, pp. 408–427. Springer, Heidelberg (2011)
4. Beisiegel, M., et al.: Service Component Architecture: Building Systems using a Service-Oriented Architecture, version 0.9. Technical report, BEA, IBM, Interface21, IONA, Oracle, SAP, Siebel, Sybase Joint Whitepaper (2005)
5. Bettini, L., Coppo, M., D'Antoni, L., De Luca, M., Dezani-Ciancaglini, M., Yoshida, N.: Global Progress in Dynamically Interleaved Multiparty Sessions. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 418–433. Springer, Heidelberg (2008)
6. Bonelli, E., Compagnoni, A.: Multipoint Session Types for a Distributed Calculus. In: Barthe, G., Fournet, C. (eds.) TGC 2007. LNCS, vol. 4912, pp. 240–256. Springer, Heidelberg (2008)
7. Boreale, M., Bruni, R., Caires, L., Nicola, R.D., Lanese, I., Loreti, M., Martins, F., Montanari, U., Ravara, A., Sangiorgi, D., Vasconcelos, V., Zavattaro, G.: SCC: a Service Centered Calculus. In: Bravetti, M., Núñez, M., Tennenholtz, M. (eds.) WS-FM 2006. LNCS, vol. 4184, pp. 38–57. Springer, Heidelberg (2006)
8. Boreale, M., Bruni, R., De Nicola, R., Loreti, M.: Sessions and Pipelines for Structured Service Programming. In: Barthe, G., de Boer, F.S. (eds.) FMOODS 2008. LNCS, vol. 5051, pp. 19–38. Springer, Heidelberg (2008)
9. Bravetti, M., Zavattaro, G.: On the Expressive Power of Process Interruption and Compensation. Mathematical Structures in Computer Science 19(3), 565–599 (2009)
10. Butler, M., Ferreira, C.: A Process Compensation Language. In: Grieskamp, W., Santen, T., Stoddart, B. (eds.) IFM 2000. LNCS, vol. 1945, pp. 61–76. Springer, Heidelberg (2000)
11. Caires, L., De Nicola, R., Pugliese, R., Vasconcelos, V., Zavattaro, G.: Core Calculi for Service-Oriented Computing. In: Hölzl, M. (ed.) SENSORIA Project. LNCS, vol. 6582, pp. 153–188. Springer, Heidelberg (2011)
12. Caires, L., Ferreira, C., Vieira, H.: A Process Calculus Analysis of Compensations. In: Kaklamanis, C., Nielson, F. (eds.) TGC 2008. LNCS, vol. 5474, pp. 87–103. Springer, Heidelberg (2009)
13. Caires, L., Vieira, H.: Conversation Types. In: Castagna, G. (ed.) ESOP 2009. LNCS, vol. 5502, pp. 285–300. Springer, Heidelberg (2009)
14. Caires, L., Vieira, H.: Conversation Types. Theoretical Computer Science 411(51-52), 4399–4440 (2010)
15. CMU-PT INTERFACES Project. Website, `http://ctp.di.fct.unl.pt/interfaces/`
16. Deniélou, P.-M., Yoshida, N.: Dynamic Multirole Session Types. In: Ball, T., Sagiv, M. (eds.) Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, pp. 435–446. ACM, New York (2011)
17. Ferreira, C., Lanese, I., Ravara, A., Vieira, H., Zavattaro, G.: Advanced Mechanisms for Service Combination and Transactions. In: Hölzl, M. (ed.) SENSORIA Project. LNCS, vol. 6582, pp. 302–325. Springer, Heidelberg (2011)
18. Fiadeiro, J.L., Lopes, A., Bocchi, L.: A Formal Approach to Service Component Architecture. In: Bravetti, M., Núñez, M., Tennenholtz, M. (eds.) WS-FM 2006. LNCS, vol. 4184, pp. 193–213. Springer, Heidelberg (2006)
19. Honda, K., Vasconcelos, V.T., Kubo, M.: Language Primitives and Type Discipline for Structured Communication-Based Programming. In: Hankin, C. (ed.) ESOP 1998. LNCS, vol. 1381, pp. 122–138. Springer, Heidelberg (1998)

20. Honda, K., Yoshida, N., Carbone, M.: Multiparty Asynchronous Session Types. In: Necula, G., Wadler, P. (eds.) Proceedings of the, 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, pp. 273–284. ACM Press, New York (2008)
21. IP Sensoria Project Website: `http://www.sensoria-ist.eu/`
22. Lanese, I., Ravara, A., Vieira, H.: Behavioral Theory for Session-Oriented Calculi. In: Hölzl, M. (ed.) SENSORIA Project. LNCS, vol. 6582, pp. 189–213. Springer, Heidelberg (2011)
23. Lanese, I., Vasconcelos, V.T., Martins, F., Ravara, A.: Disciplining Orchestration and Conversation in Service-Oriented Computing. In: 5th International Conference on Software Engineering and Formal Methods, pp. 305–314. IEEE Computer Society Press, Los Alamitos (2007)
24. Milner, R., Parrow, J., Walker, D.: A Calculus of Mobile Processes, Part I + II. Information and Computation 100(1), 1–77 (1992)
25. Sangiorgi, D., Walker, D.: The π-calculus: A Theory of Mobile Processes. Cambridge University Press, Cambridge (2001)
26. Vieira, H., Caires, L., Seco, J.: The Conversation Calculus: A Model of Service-Oriented Computation. In: Drossopoulou, S. (ed.) ESOP 2008. LNCS, vol. 4960, pp. 269–283. Springer, Heidelberg (2008)
27. Vieira, H.T.: A Calculus for Modeling and Analyzing Conversations in Service-Oriented Computing. PhD thesis, Universidade Nova de Lisboa (2010)
28. Yoshida, N., Deniélou, P.-M., Bejleri, A., Hu, R.: Parameterised Multiparty Session Types. In: Ong, C.-H.L. (ed.) FOSSACS 2010. LNCS, vol. 6014, pp. 128–145. Springer, Heidelberg (2010)

QoS Contract-Aware Reconfiguration of Component Architectures Using E-Graphs

Gabriel Tamura[1,2], Rubby Casallas[1], Anthony Cleve[2], and Laurence Duchien[2]

[1] University of Los Andes, TICSw Group, Cra. 1 N° 18A-10, Bogotá, Colombia
[2] INRIA Lille-Nord Europe, LIFL CNRS UMR 8022, University of Lille 1, France
{gabriel.tamura,anthony.cleve,laurence.duchien}@inria.fr,
rcasalla@uniandes.edu.co

Abstract. In this paper we focus on the formalization of component-based architecture self-reconfiguration as an action associated to quality-of-service (QoS) contracts violation. With this, we aim to develop on the vision of the component-based software engineering (CBSE) as a generator of software artifacts responsible for QoS contracts. This formalization, together with a definition of a QoS contract, forms the basis of the framework we propose to enable a system to preserve its QoS contracts. Our approach is built on a theory of extended graph (e-graph) rewriting as a formalism to represent QoS contracts, component-based architectural structures and architecture reconfiguration. We use a rule-based strategy for the extensible part of our framework. The reconfiguration rules are expressed as e-graph rewriting rules whose left and right hand sides can be used to encode design patterns for addressing QoS properties. These rules, given by a QoS property domain expert, are checked as *safe*, i.e., terminating and confluent, before its application by graph pattern-matching over the runtime representation of the system.

1 Introduction

In the last ten years, Component-based Software Engineering (CBSE) has evolved based on a fundamental vision of the components as a contract or obligations-responsible software artifacts [1]. On this vision, CBSE has been used as a fundamental approach for engineering software systems in a wide variety of forms. These forms include the building of systems from contract-compliant components to abstracting reflection mechanisms at the component-level (i.e., composite, component, port, connection) to support self-adaptive systems. Even though a lot of research has been conducted on how to make components guarantee contracts on individual functionality, making component-based systems to be QoS contracts-aware is another important part of the same research question: this kind of contracts constitute the base to differentiate and negotiate the quality of the service or provided functionality at the user level.

Nonetheless, providing a component-based software system with reconfiguration capabilities to preserve its QoS contracts presents several difficulties: (i) the expression of the QoS contract itself, given that it must specify the different contextual conditions on the contracted QoS property, and the corresponding guaranteeing actions to be performed in case of the QoS contract disruption [13,15]; (ii) in contraposition to functional contracts, which can be checked statically, QoS contracts are affected by global and extra-functional behaviour that must be evaluated at runtime. This evaluation requires also dynamic monitoring schemes, different to the static ones usually found in current systems [7]; (iii) several reconfiguration strategies can be used to address each desirable condition on a QoS property. These strategies are provided by different disciplines (e.g., those related to performance, reliability, availability and security), and constitute a rich knowledge base to be exploited. Nonetheless, due to their diversity of presentation in syntax and semantics, it is difficult to manage them uniformly, thus existing approaches use them as fixed subsets [2]; (iv) the reconfiguration process is required to guarantee both, the preservation of the system integrity as a component-based software system, and the correct and safe application of the reconfiguration strategy. This requirement is specially challenging if the strategies are parametrized, for instance, by using rules, still being a research issue in self-reconfiguring approaches [12].

On the treatment of software contracts several works have been proposed. Notably among them, the *design by contract* specification of the Eiffel programming language [16] and the *Web Service Level Agreement (WSLA)* initiative [15]. The design by contract theory, one of the most inspiring in the object-oriented programming paradigm, makes routines self-monitoring at compile-time by using assertions as integral parts of the source code to be checked at runtime. The violation of an assertion, such as a class invariant, is automatically managed by standard mechanisms like the *rescue* clause. The programmer must handle it appropriately to restore a consistent state. This idea was later generalized by Beugnard et al. to four types of software contracts, including those based on QoS, though not fully developed [3]. On the other side, WSLA specifies QoS contracts independent from the source code, thus involving conditions based on the actual context of execution. The WSLA includes a guaranteeing action in response to disrupted SLAs, but the semantics of this action is limited to operations such as event notification [15]. However, despite these and other many advances, the development of a well-founded theory to manage QoS contracts in component-based systems is still a challenging question.

Our goal in this paper is to formally model the *architecture reconfiguration* of a component-based (CB) system as an action performed by itself. These actions are performed in response to the disruption of QoS contracts, in the spirit of the Eiffel's *rescue* clause in object-oriented programming. By doing this, we aim to develop on the vision of the CBSE as a sound base to produce software systems enabled to automatically and safely reconfigure themselves by reconfiguring their abstract (reflection) architectures at runtime. For such structural reconfigurations, a system architect may reuse design patterns from other disciplines with the purpose of restoring QoS contracts, thus preserving them.

Our approach is built on the theory of extended graph (e-graph) rewriting proposed in [10], as a formalism to represent QoS contracts, component-based architectural structures and architecture reconfiguration. For the self-reconfiguration, we use a parametrized, *rule-based* strategy. That is, the reconfiguration possibilities are expressed as e-graph rewriting rules whose left and right hand sides can be used to encode variations of design patterns for addressing QoS properties. These rules are applied by graph pattern-matching over the system runtime e-graph representation when it is notified with events related to the violation of the corresponding properties.

The contribution of this paper is twofold. We provide (i) formal definitions for QoS contracts, CB system reflection and reconfiguration rules, in a unified framework (i.e., syntax and semantics). This allows the verification of CB structural rules of formation to be checked; and (ii) a well-founded basis for a system to manage its own reconfigurations to address the disruption of its associated QoS contracts. Once parametrized with a specific set of rules, the system can be checked as terminating (the process of rule application is guaranteed to end) and confluent (the rule application order is irrelevant and always produce the same result).

This paper is organized as follows. Section 2 presents our motivation and proposal scope. Section 3 introduces a reliable video-conference system as an example scenario to illustrate our proposal. Section 4 presents our formalization for QoS contracts-ware system reconfiguration by using e-graphs. Section 5 analyze the properties of our reconfiguration system as a result of its formalization. Section 6 compares our approach with similar proposals. Finally, Section 7 concludes the paper and anticipates future work.

2 Motivation and Scope

As defined by Oreizy et al., self-adaptive software evaluates its own behaviour at runtime and modifies itself whenever it can determine that it is not satisfying its requirements [17,20]. In their proposal, they defined the system adaptation as a cycle of four phases: (i) monitoring of context changes; (ii) analysis of these changes to decide the adaptation; (iii) planning responsive modifications over the running system; and (iv) deploying the modifications. In our proposal, we focus on the planning phase considering self-reconfiguration at the component level, triggered by sensible changes in context that affect the fulfillment of contractual QoS properties. Other component-based proposals such as COSMOS [9] and MUSIC [18] can be used for more general functionalities of context monitoring and analysis phases, meanwhile those like Fractal [4] and OSGi [21] for the component management at the deployment and execution phases.

Our motivation in this paper is to define a safe, rule-based framework to address QoS contracts violation in CB systems through the reconfiguration of the components-architecture, meaning: (i) (*rule-based reconfiguration*) the addition or removal of software components and connectors at runtime, as specified by parametrized rules given by a QoS property domain expert or a software QoS architect; (ii) (*safe-1*) these rules can be checked to be *terminating* and *confluent*, i.e., their application can be guaranteed to finish the production of the reconfiguration actions in a deterministically way. This verification is done despite the rules given being whether or not pertinent for the QoS property preservation, but correct in their definition; (iii) (*safe-2*) the QoS property domain expert is concerned only with rule specification, not with the specific procedure to apply it; (iv) (*safe-3*) once executed the reconfiguration actions into the runtime system, its CB-structural conformance can be verified.

3 Running Example

We illustrate the requirements for dynamic reconfiguration with a simplified version of a reliable mobile video-conference system (RVCS). To the user, the service is provided through a video-conference client subject to a QoS contract on its reliability. Thus, software clients are expected to be responsible for maintaining the service to the user in a "smart" way, as illustrated in Fig. 1. Note that addressing these requirements statically (e.g., with *if-then* clauses on context conditions) would not be satisfactory: as the video-conference requires bi-directionality, this would introduce synchronization issues between the client's and server's conditions, being their respective contexts not necessarily the same.

In this example, reliability is interpreted following [2], i.e., to ensure the continued availability of the video-conferencing service hosted by a corporate network. The corporate network requires all clients to access the intranet through connections guaranteeing confidentiality. Thus, even though the contract is on

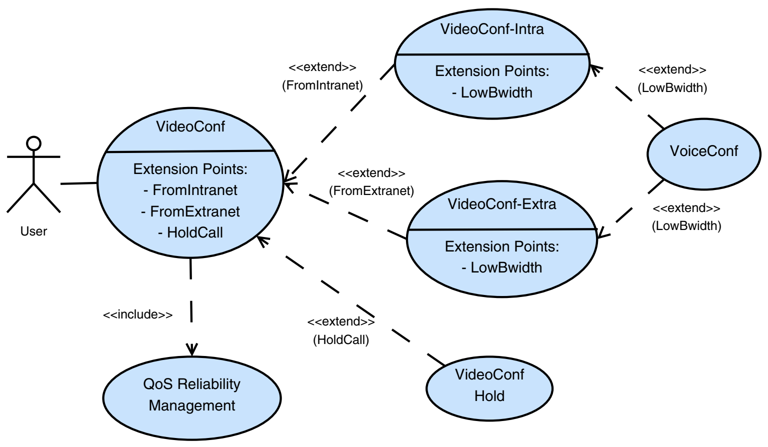


Fig. 1. Use case diagram for the requirements of the RVCS example. Connections from the intranet are considered secure, thus clear communication channels can be used. From the extranet, confidential channels are required to be configured. In case of no connection, the call must be put on hold. If the user goes into a low-bandwidth area, the system must reconfigure itself to drop the bi-directional video signals. The *QoS Reliability Management* has the responsibility of reconfiguring the system architecture to address the QoS contract violation in each case (taking into account the system's actual state) in a transparent way.

Table 1. QoS contractual conditions and corresponding service level objectives for the confidentiality property (based on access to corporate network)

Contextual Condition	Service Level Objective
CC1: Connection from Intranet	Clear Channel
CC2: Connection from Extranet	Confidential Channel
CC3: No Network Connection	Call on Hold

the QoS property of reliability, it involves two sub-properties, confidentiality and availability. On these two sub-properties, the contractual interest is on establishing the minimum levels for service acceptability (*service level objectives*), under the possible contextual conditions of system execution (cf. Tables 1 and 2).

Initially, assume the mobile user joins a video conference from her office at the corporate building, e.g., from an intranet WiFi access-point. In this state, as the contractual condition $CC1$ in Table 1 requires a clear-channel communications configuration, the system is expected to configure itself to satisfy that condition. A second system state is reached when she moves from her office to outside of the company building thus connecting through any of the available extranet wireless access-points, such as GSM or UMTS. This context change, signaled by a new contextual condition, disrupts the confidentiality contract that was being fulfilled by the actual system configuration. In this new state, according to condition $CC2$, a confidential-channel configuration on the mobile is required. The expected system behaviour is then to reconfigure itself in response to this change, in a transparent way, adopting, for instance, one of the strategies for secure multimedia transport like those defined in [23,19], thus restoring the contract. The corresponding contrary reconfiguration would apply whenever she moves back to an access-point covered by the intranet. If there are several available network access-points, a cost function should be used to choose the cheapest. Finally, whenever there is no network connection by any access-point, the call must be put on hold awaiting for automatic reconnection, just expressing that this is preferable to the alternative of dropping the service.

For illustration purposes, Table 2 establishes the minimum expected service, according to the network bandwidth, independent of the network access-point location.

Table 2. QoS contractual conditions and corresponding service level objectives for the availability property (based on network bandwidth in kbit/s)

Contextual Condition	Service Level Objective
CC4: $BandWidth \leq 12$	Call on Hold
CC5: $12 < BandWidth \leq 128$	Voice Call
CC6: $128 < BandWidth$	Voice and Video Call

4 E-Graph Modeling of QoS Contracts-Based System Reconfiguration

Given that QoS properties are dependent on system architecture, we build our proposal for making CBSE systems to be QoS contracts-responsible on a formal modeling for component-based architecture self-reconfiguration. This formalization is built on the extended theory of graph transformation given in [10].

For a CBSE system to be QoS contracts-responsible in an autonomous way, it requires (i) to have a structural representation of itself at the component level (i.e., to be reflective) [8]; (ii) to have a representation of its QoS contracts: the service level objectives for each of the contractual QoS properties, under the different contextual conditions; (iii) to be self-monitoring, that is, to identify and notify events on the contractual QoS properties violation; and (iv) to apply the architecture reconfiguration to restore the violated QoS property condition, as specified in the QoS contracts.

In Sect. 4.1 we recall the base definitions of e-graphs given in [10]; then, we use these definitions in sections 4.2 and 4.3 as a unified formalism to represent reflection structures for component-based systems and QoS contracts respectively. Finally, in Sect. 4.4 we present our proposal for architecture reconfiguration based on e-graph rewriting rules, illustrating how these defined constructs give support for reflective, autonomous and QoS contracts-based self-reconfiguring systems.

4.1 Extended Graphs: Base Definitions

Definition 1 (E-Graph). *An E-Graph is a tuple* $(V_1, V_2, E_1, E_2, E_3, (source_i, target_i)_{i=1,2,3})$, *where*

- V_1, V_2 *are sets of graph and data nodes, respectively;*
- E_1, E_2, E_3 *are sets of edges (graph, node attribution and edge attribution, respectively);*
- $source_1 : E_1 \to V_1$; $source_2 : E_2 \to V_1$; $source_3 : E_3 \to E_1$ *are the source functions for the edges; and*
- $target_1 : E_1 \to V_1$; $target_2 : E_2 \to V_2$; $target_3 : E_3 \to V_2$ *are the target functions for the edges, as depicted in Fig. 2.*

Definition 2 (E-Graph morphism). *An e-graph morphism f between e-graphs G and H, $f : G \to H$, is a tuple* $(f_{V_1}, f_{V_2}, f_{E_1}, f_{E_2}, f_{E_3})$ *where* $f_{V_i} : G_{V_i} \to H_{V_i}$ *and* $f_{E_j} : G_{E_j} \to H_{E_j}$ *for* $i = 1, 2,\ j = 1, 2, 3$, *such that f commutes with all source and target functions*[2] *(cf. Fig. 3).*

[2] Note that E-Graphs combined with E-Graph morphisms form the category *EGraphs*. See [10] for more details on this topic.

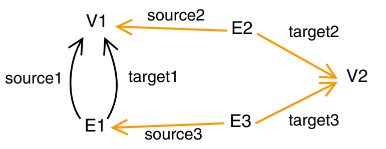


Fig. 2. E-Graph definition. An e-graph extends the usual definition of a base graph, $(V_1, E_1, source_1, target_1)$, with (i) V_2, the set of attribution nodes; (ii) E_2 and E_3, the sets of attribution edges; and (iii) the corresponding *source* and *target* functions for E_2 and E_3, used to associate the attributes for V_1 and E_1, respectively, to V_2

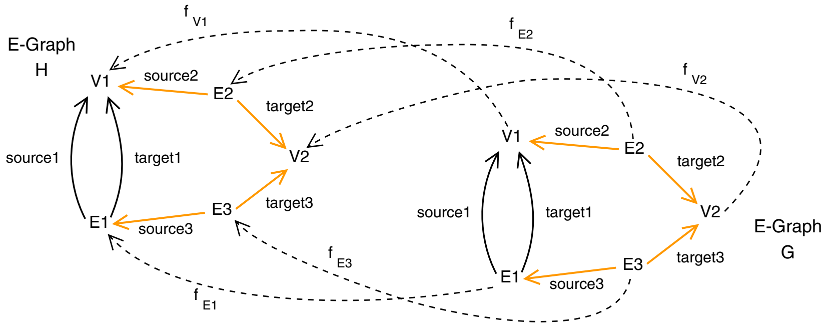


Fig. 3. E-Graph morphism illustration example f between e-graphs G and H, $f : G \rightarrow H$. E-graph morphisms are used as typing relationships between e-graphs.

4.2 System Reflection

For a system to self-reconfigure at runtime, it is required to be reflective. That is, it must be able to identify and keep track of the individual elements that are to be involved in reconfiguration operations [8]. In our case, the reflection structure is defined on a component-based structure that comprises the CBSE component, port, port type and connector elements. Composites are abstracted as components, as we address structural reconfiguration at the system level.

Definition 3 (Component-Based Structure - CBS). *The component-based structure, CBS, is the tuple $(G, DSig)$, where*

- *$DSig$ is a data signature over the disjoint union $String + PortRole$ and $PortRole = \{Provided, Required\}$, with the usual CBSE interpretations;*
- *G is the e-graph $(V_1, V_2, E_1, E_2, E_3, (source_i, target_i)_{i=1,2,3})$ such that $V_1 = \{SReflection, Component, Port, PortType, Connector\}$; each of the data nodes is named after its corresponding sort in $DSig$, $V_2 = String{+}PortRole$; $E_1 = \{component, port, provided, required, type\}$, $E_2 = \{cname, pname, ptype, role, c.QoSProvision, p.QoSProvision, ct.QoSProvision\}$, $E_3 = \{\}$; and the functions $(source_i, target_i)_{i=1,2,3}$ are defined as depicted in Fig. 4.*

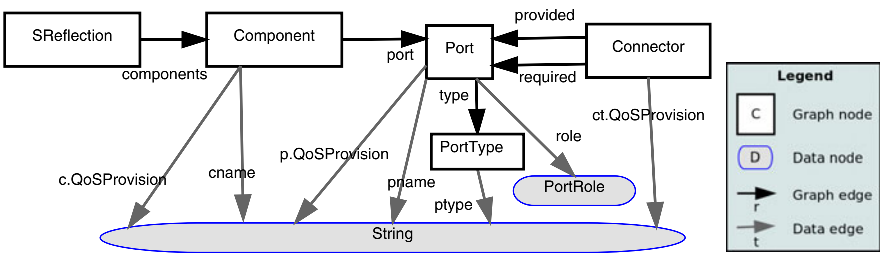


Fig. 4. The Component-Based Structure, *CBS*, is defined as an e-graph where each of the graph nodes represents each of the CBSE elements. The graph edges correspond to the relationships among these elements, meanwhile the data edges, to their corresponding attributes; *QoSProvision* is a special attribute for components, ports and connectors to express that they warrant a particular QoS condition, such as providing a secure connection to a network. The data nodes represent the types of these attributes.

Definition 4 (Component-Based System Reflection). *Given S the computational state of a running component-based system, its corresponding reflection state, R_S, is defined as $R_S = (G, f_S, t)$, where G is the e-graph that represents S through the one-to-one function $f_S : S \to G$, and t is an e-graph morphism $t : G \to CBS$.*

That is, S represents the state of each of the system components, ports and connectors as maintained in a component platform such as FRACTAL or OSGi. The feasibility of f_S results from Def. 3 (CBS) and the e-graph morphism t. $R_S.Component$ denotes the set of components in R_S, i.e., $R_S.Component = \{c | c \in G_{V1} \wedge t_{V1}(c) = Component\}$ (analogously for the other CBS elements). The purpose of f_S is to map the system architecture into the e-graphs domain, in which the architecture reconfiguration is operated. Once reconfigured, we use f_S^{-1} to perform the reconfiguration back in the actual runtime component-based system.

Example 1 (Video Conference System). Figures 5 and 6 illustrate, respectively, the runtime component-based system structure of our video-conference example and its corresponding system reflection state, when configured to be connected from the intranet (i.e., with a clear-channel connection).

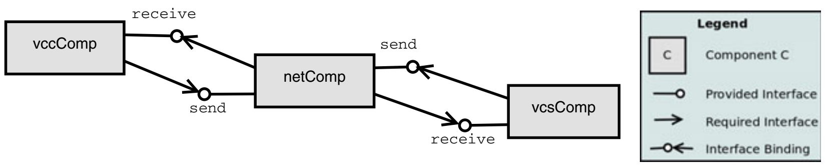


Fig. 5. Runtime system structure for Ex. 1 with a clear channel connection

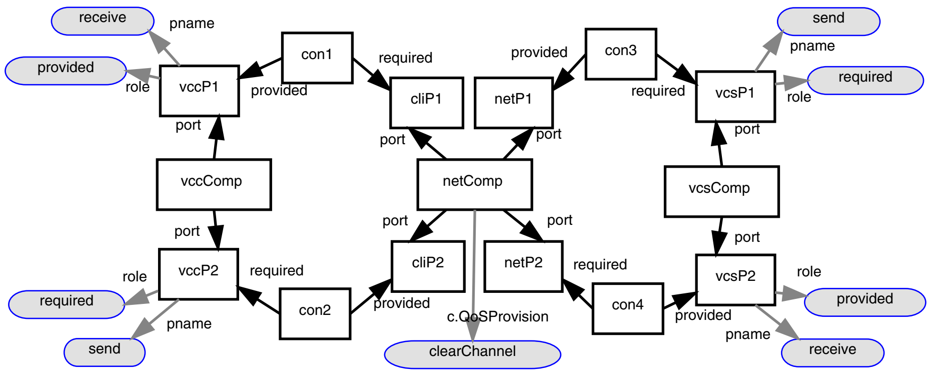


Fig. 6. Runtime system reflection structure, in e-graph notation, for the runtime system of Fig. 5 (i.e., when connected from the intranet). The *netComp* component, providing a network connection, is responsible for maintaining a *clearChannel* connection, as expressed by its *c.QoSProvision* attribute. Further details omitted for space.

The components of Fig. 5 are represented in Fig. 6 as exactly the video conference client with its network connection (*vccComp* and *netComp*) and the server (*vcsComp*). The other elements in represent their ports ($vccP1$, $vccP2$ and so on), and these port's actual connections ($con1$, $con2$ and so on).

4.3 QoS Contracts

A *QoS contract* is a specification of the guarantees on QoS properties under specific conditions for a given functionality, as offered by a system or service provider to *any* of its potential clients [14,3]. In this sense, a QoS contract is an invariant that a system must preserve, for instance, by restoring it in case of its violation. The evaluation of the invariant validity must be performed at runtime, given that it depends on measurements from the actual context of execution, such as response time, throughput, and security level on network access location; therefore, the QoS property condition must be monitored and the system must act upon its violation in order to have the possibility of restoring it opportunely.

For a system to address its QoS contracts' violation, it must incorporate and manage these contracts internally. Given our formal modeling of a component-based system as a realization of system reflection, we use the same formal framework to define QoS contracts as a manageable part of the system.

Definition 5 (QoS Contract). *Given QoSDSig the usual data signature over the disjoint union String + Boolean, a QoS contract is a tuple* (C, ct)*, where*

– *C is an e-graph representing the contract instance;*
– *ct is an e-graph morphism* $ct : C \to Q$*, where Q is the e-graph reference definition for QoS contracts,* $(V_1, V_2, E_1, E_2, E_3, (source_i, target_i)_{i=1,2,3})$ *such*

that $V_1 = \{QoSContract, QoSProperty, QoSMonitor, QoSGuarantor, SLOObligation, QoSRule\}$*; each of the data nodes is named after its corresponding sort in* $QoSDSig$*,* $V_2 = String + Boolean$*;* $E_1 = \{property, obligation, monitor, guarantor, ruleSet\}$, $E_2 = \{gname, pname, mname, rname, SLOPredicate, contextCondition, isActive\}$, $E_3 = \{\}$*; and the functions* $(source_i, target_i)_{i=1,2,3}$ *are defined as depicted in Fig. 7.*

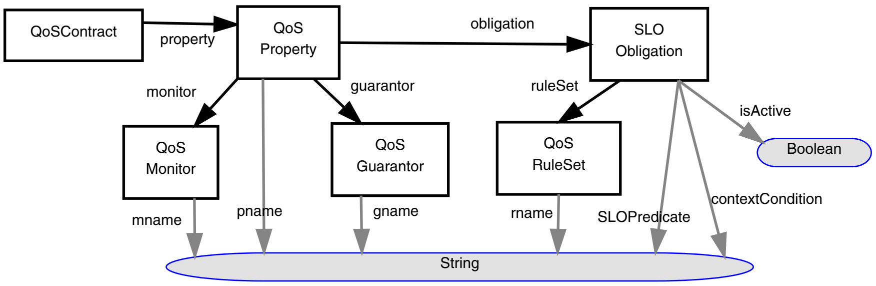


Fig. 7. E-graph reference definition for QoS contracts. Following [22] and [13], we define a QoS contract on QoS properties (*QoSProperty*). For each property, a set of service level objective obligations (*SLOObligation*) is specified. An SLO obligation establishes (i) the possible context conditions (*contextCondition*) of system execution; (ii) the SLO to be fulfilled (*SLOPredicate*) under these conditions; and (iii) a guaranteeing reconfiguration rule set (*QoSRuleSet*) to be applied in case of SLO violation. The *QoSGuarantor* refers to the system element that should provide the contracted functionality under the specified SLO obligations. The identification and notification of context changes and of SLOs violations is a responsibility of the *QoSMonitor*.

Example 2 (QoS Contract on Confidentiality). Table 3 illustrates the contract on the QoS property of confidentiality for our video-conference system example. The corresponding e-graph representation is given in Fig. 8.

Table 3. QoS contract example on confidentiality for the video-conference system

System Obligations		
Context Condition	Service Level Objective	Guaranteeing Rule Set
1: $conn\_from\_intranet$	$clearChannel$	R.clearChannel
2: $conn\_from\_extranet$	$confidentChannel$	R.confidentChannel
3: $no\_network\_conn$	$localCache$	R.localCache
Responsibilities		
- System Guarantor:	$System.netComp$[a]	
- Context Monitor:	$System.netComp\_AccessPointProbe$[b]	

[a] The system component providing the network connection under the required QoS conditions.

[b] The designated component to check changes on the system network connection's access points and corresponding confidentiality violations.

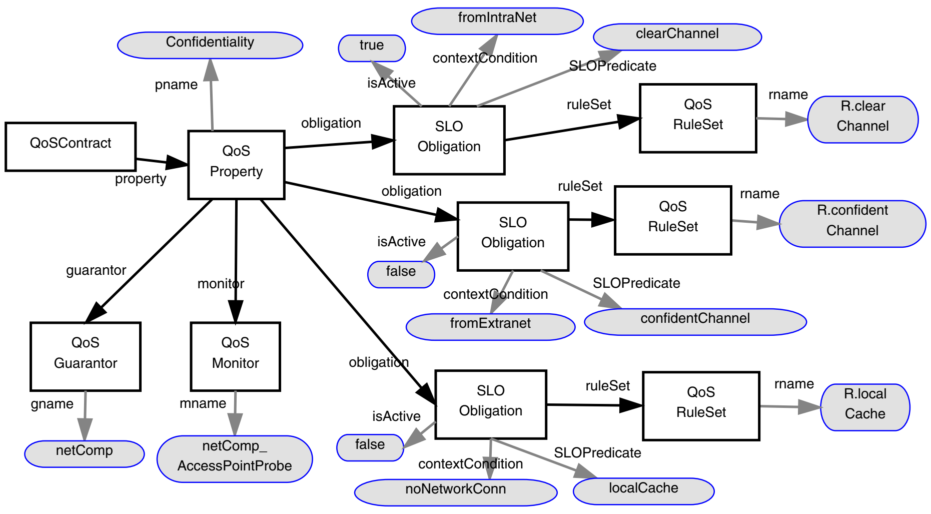


Fig. 8. QoS Contract, in e-graph notation, for the video-conference example. This contract specifies the *netComp* component (cf. Fig. 6) as the *QoSGuarantor*, and an *AccessPointProbe* on this component as the *QoSMonitor* for the confidentiality QoS property. This monitor is used by the system to continually check the changes in the context conditions and violations of the actual SLO. In our example, the initial context condition is a connection *fromIntranet*, and the corresponding SLO is to maintain a *clearChannel*. A context change in connection *fromIntranet* to *fromExtranet* triggers the application of the respective reconfiguration rule set, *R.confidentChannel*. Then, the new context condition would be activated (connection *fromExtranet*). (cf. Tab. 3).

4.4 Component-Based Architecture Reconfiguration Modeling

Having formalized the structural parts of a system in terms of e-graphs, we define the runtime software architecture reconfiguration as an e-graph transformation system. The definition of this reconfiguration system is based on a definition of a reconfiguration rule.

Definition 6 (Reconfiguration Rule). *A reconfiguration rule, p, is a tuple $(L, K, R, l, r, lt, kt, rt)$, where L (left hand side), K (left-right gluing), and R (right hand side) are e-graphs, and l, r, lt, kt, rt are graph morphisms, abbreviated, $p = (L \xleftarrow{l} K \xrightarrow{r} R)$, and $lt : L \to CBS$, $kt : K \to CBS$ and $rt : R \to CBS$. p is said to reconfigure L into R.*

Conceptually, a reconfiguration rule specifies a strategy to address conditions on QoS properties. Thus, for each guaranteeing rule set specified in the QoS contract, associated to a context condition on a given QoS property, the user can encode architectural patterns that address that condition in the left and right hand sides of the rules. Different left hand sides for a similar right hand side in a rule set for a given condition are possible, since the system structures depend on the different context conditions. All left-hand sides of rules in a rule-set are named after that rule-set name. In the scenario of our example, for instance, it is possible to change to a connection from the extranet by moving either from the intranet or from a state with no network connection. Each of these two conditions requires its own system structure, namely, a clear-channel or a local-cache structure, respectively.

Example 3 (Reconfiguration rule). The QoS contract on confidentiality for our video-conference example specifies a guaranteeing set of reconfiguration rules, *R.confidentChannel*, to address the context change when the user moves to the extranet and the contract is violated. Figure 9 illustrates the rule (in that set) that applies when the user is moving from the intranet.

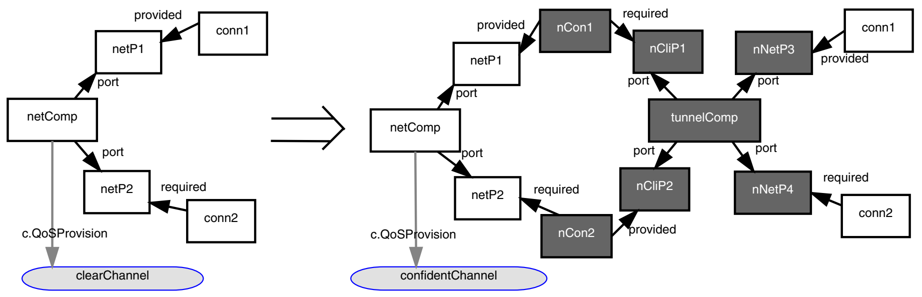


Fig. 9. The *R.confidentChannel* reconfiguration rule, in e-graph notation, that applies when moving from an intranet network connection to an extranet connection. The left-hand side (LHS) of the rule is used by a pattern-matching algorithm to find a component *netComp* in the system, such that it supports a *clearChannel* as SLO obligation (by the *c.QoSProvision* attribute). The right-hand side (RHS) specifies that (i) the matched components by the LHS must be kept with their corresponding connectors, except those for $conn1$ and $conn2$; (ii) the dark elements must be configured and deployed to provide a tunneled (i.e., confident) channel for the data; (iii) the new ports $nCon1, nCon2$ must be connected to the previously existing ports $netP1, netP2$, and $conn1, conn2$ reconnected to the new ports $nNetP3, nNetP4$, respectively; and (iv) the *c.QoSProvision* attribute of *netComp* must be updated as provisioning a *confidentChannel*. For clarity, the left-right gluing K and graph morphisms l, r, lt, kt, rt are omitted in this figure; K, l, r would correlate each of the corresponding non-dark elements in the RHS with their LHS's counterparts.

Definition 7 (Reconfiguration System). *A component-based reconfiguration system is a tuple* $(DSig, CBS, S, C, P)$*, where* $DSig$ *is a suitable data type signature for component-based systems,* CBS *the component-based structure definition (Def. 3), and* S *the structure of the system to reconfigure in its initial state,* C *a QoS contract, and* P *a set of reconfiguration rules (with* S*,* C *and* P *according to Def. 4, 5 and 6, respectively) in which:*

1. *(When to reconfigure) A system reconfiguration is triggered whenever the QoS-Monitor specified in the contract C, C.monitor, notifies of an event that violates the actual SLO (C.property.obligation.SLOPredicate). This event signals that a new context condition, related to another C.property.obligation. contextCondition, is currently in force. Associated to this new context condition, the contract specifies the corresponding SLO and guaranteeing reconfiguration rule set $P = C.property.obligation.ruleSet$.*
2. *(How, Where and What to reconfigure) The identified rule set P is applied to the system reflection structure R_S of S. That is, for each reconfiguration rule $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$ in P, and morphism $m : L \rightarrow G$ (called a match of the left-hand side of p, L, in G), we identify a direct reconfiguration $G \overset{p,m}{\Rightarrow} H$ as an e-graph transformation of G into H, as specified by the reconfiguration rule p, of L into R, according to Def. 6.*
3. *A one-step system reconfiguration is a sequence of direct transformations $G_0 \Rightarrow G_1 \Rightarrow \ldots \Rightarrow G_n$, written $G_0 \overset{*}{\Rightarrow} G_n$, until no more rules in P can be applied.*
4. *The system reconfiguration finishes with a new e-graph reflection system structure, R'_S. The list of actions to reconfigure R_S into R'_S can then be applied to the actual runtime system through f_S^{-1}, according to Def. 4.*

Example 4 (System reconfiguration). Figure 10 illustrates the reconfigured runtime system structure having applied the reconfiguration rule of Example 3 (to be used when the network connection changes from the intranet to the extranet).

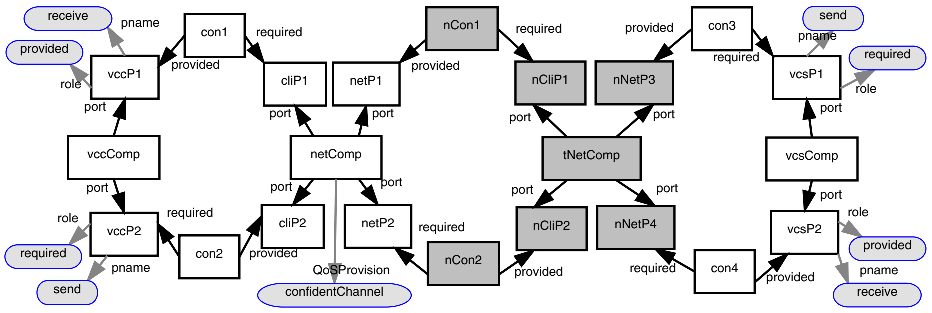


Fig. 10. Reconfigured system architecture in e-graph notation. This new system structure fulfills the SLO (*confidentChannel*) for the new context condition (network connection *fromExtranet*), as specified in the contract illustrated in Fig. 8. The added components are highlighted (shaded). Further details omitted for clarity.

5 QoS Contracts-Based Reconfiguration Properties

In this section we analyze the properties of our proposed reconfiguration system as a result of the formalization presented in the previous section.

5.1 Component-Based Structural Compliance

Definition 8 (Full CB-Structural Compliance). *A runtime system reflection structure, RS, is full CB-structural compliant if it is a component-based structure (i.e., if there exists a graph morphism $t : RS \to CBS$), and the following conditions hold* [1]*:*

1. $\forall c(c \in RS.Connector \implies \exists p, q(p, q \in RS.Port \implies c.provided = p \wedge c.required = q \wedge c.provided \neq c.required))$*: the ports referenced by the provided and required attributes must be different in every connector.*
2. $\forall p((p \in RS.Port \wedge p.role = Required) \implies \exists c(c \in RS.Connector(c.required = p)))$*: all required ports must be connected.*
3. $\forall c1, c2(c1, c2 \in RS.Connector \implies ((c1.name = c2.name \wedge c1.provided = c2.provided \wedge c1.required = c2.required) \implies c1 = c2)$*: every connector must connect different elements.*

The verifiability of full CB-structural compliance obviously results from the structural definitions 3 and 4 of our reconfiguration system proposal. Even though it would be desirable to statically check that reconfiguration rules produce only full CB-structural compliant systems, this would require more constraints on the reconfiguration rules.

Example 5 (Full CB-structural compliance). The system reflection structures of Fig. 6 and Fig. 10 are full CB-structural compliant, as it is straightforward to verify that the corresponding conditions hold on them.

5.2 Termination and Confluence of the System Reconfiguration

In [10] the *Local Church-Rosser, Parallelism* and *Concurrency* theorems, which hold for graph rewriting, are proved as valid also for typed attributed graph transformation systems. In this section, we show that the one-step system reconfiguration (i.e., $G_0 \overset{*}{\Rightarrow} G_n$ in Def. 7) of our component-based reconfiguration system is reducible to a typed attributed graph transformation system. Therefore, those theorems are also valid for our reconfiguration system.

Theorem 1 (Reducibility of One-Step System Reconfiguration). *Let CBR be a component-based reconfiguration system. A one-step component-based (CB) system reconfiguration, $CBSR$, in CBR, is reducible to a typed attributed graph transformation system, $TAGTS$.*

Proof. According to Def. 7, a component-based reconfiguration system is a tuple $(DSig, CBS, S, C, P)$. Of these elements, for one-step system reconfiguration (i.e., $G_0 \overset{*}{\Rightarrow} G_n$), the data signature, $DSig$, the component-based structure definition, CBS, and the QoS contract, C, are unchanged. Therefore, in a one-step system reconfiguration these elements can be omitted, depending only on the system reflection structure, S, and the set of reconfiguration rules, P. Given that

[1] Multiplicity constraints, as defined as usual in CBSE, are omitted for space.

1. a CB system reflection structure is a tuple (G, f_S, t), where G is the e-graph that represents a system S through the one-to-one function $f_S : S \rightarrow G$, and t is an e-graph morphism $t : G \rightarrow CBS$. In the one-step system reconfiguration, f_S also is unchanged and CBS is a type e-graph for G, G attributed with the data signature $DSig$;
2. a typed attributed graph is a tuple (AG, u), where AG is an attributed graph over a data signature $TAGDSig$, and u is an attributed graph morphism, $u : AG \rightarrow ATG$, where ATG is a type graph:
3. a CB reconfiguration rule, p, is a tuple $(L, K, R, l, r, lt, kt, rt)$, $p = (L \xleftarrow{l} K \xrightarrow{r} R)$, and $lt : L \rightarrow CBS$, $kt : K \rightarrow CBS$ and $rt : R \rightarrow CBS$;
4. the typed attributed graph transformation rules are graph rewriting productions $q = (X \xleftarrow{x} Y \xrightarrow{y} Z)$, X, Y, Z graphs; and
5. both, the system reflection structure and the typed attributed graph are based on the same e-graph definition,

a one-step system reconfiguration, $CBSR$, can be reduced to a typed attributed graph transformation system, $TAGTS$, by making $TAGDSIG = DSig$, $AG = G$ and $ATG = CBS$. The $TAGTS$ set of transformation rules can be defined as the set of CB reconfiguration rules without the lt, kt, rt morphisms, given that, once defined the CB reconfiguration rules, these morphisms are no longer required. $\square$

As a result, the *Local Church-Rosser, Parallelism* and *Concurrency* theorems can then be used with critical pair checking in a particular set of reconfiguration rules, and determine if the one-step system reconfigurations in our reconfiguration system is terminating and confluent. This verification ensures the reliability of the reconfiguration process and frees a system architect of being aware of (i) rule dependencies that may cause deadlocks in the reconfiguration; and of (ii) the rule application order and the specific procedure to perform the reconfiguration itself.

5.3 Stabilization and Exception in the Reconfiguration Process

Given that the reconfiguration rules in our proposal are specified by the user, our reconfiguration system must also consider exceptional cases. These cases correspond to two contract-unfulfilled states, namely the unstable and the exception. The unstable state is reached when a plausible reconfiguration rule has been found and applied in the system reflection structure, but its effect has not been enough to restore the contract validity. Operationally, in this state the user must be notified about the inefficacy of the rules specified in the contract, after applying the rules a given number of times. On the other side, the state of exception is reached when the reconfiguration system has not been able to find a matching rule to apply in the running system reflection structure. In this case, the user must be notified about the context condition under which the system reflection structure has no corresponding reconfiguration rule, as specified in the contract.

6 Related Work

Software contracts can be seen as a form of property preservation, being this is a recurrent problem in computer science. This problem has been addressed by different communities with different approaches, being a fundamental characteristic of mature engineering disciplines [1]. Our work has been inspired by the general framework approach of some of these proposals, addressing QoS contracts violation through system reconfiguration in component-based systems.

At least in abstract, many of these proposals follow the *rescue* clause idea of the Eiffel's design by contract theory [16]. For example, even though not on the CBSE nor addressing QoS contracts, but on the formal-based self-healing properties preservation side, in [11] Ehrig et al. used algebraic graph transformations for the static analysis and verification of specific properties. Their proposal use a fixed set of particular transformation rules to be applied in response to system failures, thus the self-healing properties are proven with them. Our proposal differs to theirs in that we want to provide a general framework, in the context of component-based software engineering, to be parametrized with reconfiguration rules given by the user; this means that they can prove specific properties, meanwhile we provide tools to the user for checking general properties. Another approach, yet non-formal, aiming at preserving system structural properties in software reconfiguration is the proposed by Hnětynka and Plášil in [12]. Their approach limit the system reconfigurations to those matching three specific *reconfiguration patterns* in order to avoid the dynamic reconfiguration to introduce system architecture inconsistencies.

On the treatment of contracts, in [6] Chang and Collet focuses on the problem of combining low-level properties of individual components to obtain system-level properties as a support for contract negotiation. Their approach identifies then compositional patterns for non-functional properties. On another side, Cansado et al. propose in [5] a formal framework for component-based structural reconfiguration and gives a formal definition of behavioural contract. Their approach is based on a labeled transition system as a formalism to unify behavioural adaptation and determine if a reconfiguration can be performed. Our proposal, even though also address system-level contracts as the two above mentioned, differs to those in that we are interested in the related problems of system architecture and the dependencies on the execution context, meanwhile those deal with more low-level component problems of property composability and interface adaptability, respectively.

7 Conclusions

The main challenge we face in this paper is how to make component-based systems QoS-contracts responsible under varying conditions of context system execution.

In order to face this challenge, we propose a formal approach based on *e-graphs* for system reflection modeling, QoS contract modeling and system architecture reconfiguration. With these definitions, we prove that the one-step system reconfiguration of our component-based reconfiguration system is reducible to a typed attributed graph transformation system. In [10] the *Local Church-Rosser, Parallelism* and *Concurrency* theorems are proved for typed attributed graph transformation systems. Therefore, the adoption of e-graphs to build our component-based transformation system represents three important benefits, as it allows us to: (i) take advantage of the properties of termination and confluence that these theorems allow to check, as a sound strategy for the development of *rule-based*, dynamic, autonomous and self-reconfiguring systems; (ii) provide a rich expressive notation by combining and exploiting graph visual presentations with graph-based pattern-matching; and (iii) benefit from the existing catalogs of design patterns that target different architecture and QoS concerns, as far as the users encode them as reconfiguration rules. In this latter case, our approach enables users to effectively reuse these software design artifacts to enforce particular QoS attribute conditions. For this, however, a more legible and usable concrete syntax should be developed, with automated tools to assist the user in the writing of reconfiguration rules in a more familiar notation such as the used in component-based specifications.

Our formal framework can be used thus to develop and implement rule-based systems in automated and safe ways, being them QoS contracts responsible. With these systems, a user is enabled to define her own rules while freeing her of being aware of the rule application order and of the details of the specific procedure to apply them. For this, and as a result of the formal definition of the QoS contract, component-based systems are enabled as self-monitoring. To this respect, QoS addressing proposals usually detect and manage contract violation either at a coarse-grained, system resources level or at the fined-grained component interfaces level. Our approach is an intermediate proposal, as it takes into account the software components but at the architecture level. Thus, the conditions on QoS properties that we can address can be measured from system context components, and the corrective actions in response to their violation are also at the component-architecture reconfiguration. Nonetheless, from a general point of view, it is possible to formalize in our proposal the global behaviour of the reconfiguration system, defining more precisely the meaning of the contract-unfulfilled states of un-stability and exception, for instance using ideas from process algebras. As future work our plan is (i) to continue the development of our formal framework to form a comprehensive theory for the treatment of QoS contracts in component-based software systems; and (ii) implement it and apply it in representative cases of study to have a better understanding of the different kind of properties that the engineering of self-adaptive software systems must address.

Acknowledgments. This work was funded in part by the Icesi University (Cali, Colombia), the Ministry of Higher Education and Research of Nord-Pas de Calais Regional Council and FEDER under Contrat de Projets Etat Region (CPER) 2007-2013, and during the tenure of an ERCIM "*Alain Bensoussan*" Fellowship by the third author.

References

1. Bachmann, F., Bass, L., Buhman, C., Comella-Dorda, S., Long, F., Robert, J., Seacord, R., Wallnau, K.: Technical Concepts of Component-Based Software Engineering. Vol. 2. Technical Report CMU/SEI-2000-TR-008, CMU/SEI (2000)
2. Barbacci, M., Klein, M.H., Longstaff, T.A., Weinstock, C.B.: Quality attributes. Technical Report CMU/SEI-95-TR-021, CMU/SEI (1995)
3. Beugnard, A., Jézéquel, J.M., Plouzeau, N., Watkins, D.: Making components contract aware. IEEE Computer 32(7), 38–45 (1999)
4. Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., Stefani, J.B.: The fractal component model and its support in java: Experiences with auto-adaptive and reconfigurable systems. Softw. Pract. & Exper. 36(11-12), 1257–1284 (2006)
5. Cansado, A., Canal, C., Salaün, G., Cubo, J.: A formal framework for structural reconfiguration of components under behavioural adaptation. In: Procs. of the 6th Intl. Workshop FACS (2009); ENTCS 263(1), 95 – 110 (2010)
6. Chang, H., Collet, P.: Compositional patterns of non-functional properties for contract negotiation. JSW 2(2), 52–63 (2007)
7. Chang, H., Collet, P.: Patterns for integrating and exploiting some non-functional properties in hierarchical software components. In: Procs. of the 14th IEEE Intl. Conference and Workshops on the ECBS 2007, pp. 83–92. IEEE CS, Los Alamitos (2007)
8. Cheng, B.H., Lemos, R., Giese, H., Inverardi, P., Magee, J., Andersson, J., Becker, B., Bencomo, N., Brun, Y., Cukic, B., Marzo Serugendo, G., Dustdar, S., Finkelstein, A., Gacek, C., Geihs, K., Grassi, V., Karsai, G., Kienle, H.M., Kramer, J., Litoiu, M., Malek, S., Mirandola, R., Müller, H.A., Park, S., Shaw, M., Tichy, M., Tivoli, M., Weyns, D., Whittle, J.: Software engineering for self-adaptive systems: A research roadmap. In: Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Software Engineering for Self-Adaptive Systems. LNCS, vol. 5525, pp. 1–26. Springer, Heidelberg (2009)
9. Conan, D., Rouvoy, R., Seinturier, L.: Scalable processing of context information with COSMOS. In: Indulska, J., Raymond, K. (eds.) DAIS 2007. LNCS, vol. 4531, pp. 210–224. Springer, Heidelberg (2007)
10. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Springer-Verlag New York, Inc., Heidelberg (2009)
11. Ehrig, H., Ermel, C., Runge, O., Bucchiarone, A., Pelliccione, P.: Formal analysis and verification of self-healing systems. In: Rosenblum, D.S., Taentzer, G. (eds.) FASE 2010. LNCS, vol. 6013, pp. 139–153. Springer, Heidelberg (2010)
12. Hnětynka, P., Plášil, F.: Dynamic reconfiguration and access to services in hierarchical component models. In: Gorton, I., Heineman, G.T., Crnković, I., Schmidt, H.W., Stafford, J.A., Ren, X.-M., Wallnau, K. (eds.) CBSE 2006. LNCS, vol. 4063, pp. 352–359. Springer, Heidelberg (2006)
13. Keller, A., Ludwig, H.: The wsla framework: Specifying and monitoring service level agreements for web services. J. Netw. Syst. Manage. 11(1), 57–81 (2003)
14. Krakowiak, S.: Middleware architecture with patterns and frameworks (2009), `http://sardes.inrialpes.fr/~krakowia/MW-Book/`
15. Ludwig, H., Keller, A., Dan, A., King, R.P., Franck, R.: Web Service Level Agreement (WSLA) Language Specification(2003), IBM Available Specification
16. Meyer, B.: Applying "Design by Contract". Computer 25(10), 40–51 (1992)
17. Oreizy, P., Gorlick, M.M., Taylor, R.N., Heimbigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D.S., Wolf, A.L.: An architecture-based approach to self-adaptive software. IEEE Intelligent Systems 14(3), 54–62 (1999)

18. Paspallis, N., Rouvoy, R., Barone, P., Papadopoulos, G.A., Eliassen, F., Mamelli, A.: A pluggable and reconfigurable architecture for a context-aware enabling middleware system. In: Chung, S. (ed.) OTM 2008, Part I. LNCS, vol. 5331, pp. 553–570. Springer, Heidelberg (2008)
19. Ramachandran, J.: Designing Security Architecture Solutions. John Wiley & Sons, Inc., New York (2002)
20. Taylor, R.N., Medvidovic, N., Oreizy, P.: Architectural styles for runtime software adaptation. In: WICSA/ECSA 2009, pp. 171–180. IEEE, Los Alamitos (2009)
21. The OSGi Alliance: OSGi Service Platform Core Specification Release 4. Tech. rep., The OSGi Alliance (June 2009), http://www.osgi.org/Download/Release4V42, oSGi Available Specification
22. Tran, V.X., Tsuji, H.: A survey and analysis on semantics in qos for web services. In: Intl. Conf. on Advanced Information Networking and Apps., pp. 379–385 (2009)
23. Zeng, W., Zhuang, X., Lan, J.: Network friendly media security: Rationales, solutions, and open issues. In: Procs. of the 2004 Intl. Conf. on Image Processing (ICIP), pp. 565–568. IEEE, Los Alamitos (2004)

Monitoring Method Call Sequences Using Annotations*

B. Nobakht[1], M.M. Bonsangue[1,2], F.S. de Boer[1,2], and S. de Gouw[1,2]

[1] Leiden Institute of Advanced Computing,
Universiteit Leiden, Niels Bohrweg 1, 2333 CA Leiden, The Netherlands
{bnobakht,marcello}@liacs.nl
[2] Centrum Wiskunde and Informatica,
Science Park 123, 1098 XG Amsterdam, The Netherlands
{frb,cdegouw}@cwi.nl

Abstract. In this paper we introduce JMSeq, a Java-based tool for the specification and runtime verification via monitoring of sequences of possibly nested method calls. JMSeq provides a simple but expressive way to specify the sequential execution of a Java program using code annotations via user-given sequences of methods calls. Similar to many monitoring-oriented environments, verification in JMSeq is done at runtime, but differently from all other approaches based on aspect-oriented programming, JMSeq does not use code instrumentation, and therefore is suitable for component-based software verification.

Keywords: Object monitoring, run-time verification, method call sequence specification, code annotation, component-based testing, communication traces.

1 Introduction

Testing and, more recently, monitoring are two established approaches to the verification of large complex systems. Testing is generally used to validate the results of each step in the software life cycle against the expected ones. More recently, monitoring-oriented programming has emerged as a formal branch of testing suitable to validate runtime data collected during system execution [13]. While in ordinary testing the software system under test must be stimulated so to reproduce an expected behavior, in monitoring the actual behavior is observed and analyzed a posteriori with respect to some specification.

Component based software engineering advocates the construction of software by gluing together prefabricated components [27]. Because the code of the components is not always available, automatic code insertion is not possible, thus posing new challenges to testing and monitoring of the final product. An alternative to the white box view of monitoring is given by automatic generation

* This research is partly funded by the EU project FP7-231620 HATS: Highly Adaptable and Trustworthy Software using Formal Models (`http://www.hats-project.eu/`)

of wrappers for monitoring the external behavior of third-party components. A wrapper is automatically generated for every component with tracking code to form an observable component in the black box view. As for the automatic code insertion, formal specifications are separated from source code, but the complexity of generating the wrappers can be very high, as the code relative to a single specification must be distributed among several wrappers. The approach is thus not flexible and easily reusable. Basically, there are no other systematic methods and technologies available to control and monitor the external behaviors of the components according to a test specification.

In this paper we consider components as compiled Java packages that are annotated with specification of the internal object behaviors by means of sequences of outgoing method calls from the component and incoming return messages to the component. Component designers can use the standard Java 5 Annotations [3] to specify the intended behavior of the component as well as the action to be taken in case of failure. We express the intended behavior through a set of execution traces[1]. To check conformance of the execution trace with respect to its specification, monitors are automatically synthesized from specified properties and integrated into the original system to check its dynamic behaviors during execution. The approach is therefore simple, modular and flexible to use. It does not assume that the component source code is available, but it requires the addition of annotations either at design process or the testing phases, similar to what happens in the proof-carrying code approach [25]. The runtime verification and monitoring framework we implement is based on the Java Platform Debugger API (JPDA) [4]. Our method is thus complementing the approach taken in JavaMOP [14], where Java programs are monitored and verified at runtime by means of code instrumentation. JMSeq can be used to complement other existing testing and verification frameworks to add capabilities for the developer. For instance, one can use JML [22] to specify the properties of the data flow and use JMSeq to specify the properties of the control flow.

The rest of the paper is organized as follows. In Section 2 we discuss the problem of monitoring and testing component based systems. Then, in Section 3 we present a language for specifying sequences of method calls that are annotating the interfaces or the code of a program, as explained in Section 4. These annotations are used by the JMSeq framework, introduced in Section 5, to monitor the program. In Section 6, we discuss related work including a more detailed comparison with JavaMOP [14], another monitoring framework. Finally, we conclude in Section 7 and also discuss possible future work.

2 Monitoring Component Based Systems

Monitoring refers to the activity of tracking observable information from an execution of a system with the goal of checking it against the constraints imposed by the system specification. The observable information of the monitored system

[1] By execution traces, we mean the sequence of method calls and method returns, not sequences of states.

typically includes behaviors, input and output, but may also contain quantitative information. A monitoring framework consists of a monitor that extracts the observable behavior from a program execution, and a controller that checks the behavior against a set of specifications. In the case that an execution violates the constraints imposed by the specification, corrective actions can be taken.

What can be monitored depends, of course, on what can be observed in a system. When the application source code is available, its code can be instrumented to receive informative messages about the execution of an application at run time. New code is inserted into the original code, preserving the original logic of the application; yet, the extra code is essential for the management of monitoring and verification mechanisms. For example JavaMOP [14] uses AspectJ to inject monitors into the original code. Also, the JML run-time assertion checker has been implemented using aspect-oriented programming [26].

As programs have become larger and more complex, encapsulation and hiding techniques have become more important. Component based software construction has emerged as a viable solution for handling the complexity of software. Components implement one or more interfaces describing the services they provide and require. Usually, the interfaces are described only in terms of a signature and the source code of a component is not available. During the integration phase, when components are composed into a system, it is thus very difficult for the developer to check if the behavior of the system conforms to its specification. In fact, the absence of source code implies that we cannot instrument it, and therefore current run-time verification techniques cannot be used.

In this paper we present JMSeq, a runtime verification framework that is not based on code instrumentation, but rather on *code annotation*. Code annotation has become increasingly popular in the past few years, especially because of its effectiveness in integrating formal techniques for verification with programming. Essentially, annotations are code segments that are compiled but do not provide any logic or business in the program; yet, they indirectly affect the program execution based on the additional information they add to the running code.

Annotations are different from documentation tags as originally used by JML to specify assertions for verification of Java code [22]. In fact documentation tags are not compiled, and thus are highly dependent on the presence of the source code. The success of modern testing frameworks such as JUnit [5] advocated for development of services based on annotations for the recent versions of the Java language. JMSeq uses Java annotations to provide a way for the programmer to specify for each method a set of execution traces representing an abstraction of the intended behavior, or protocol, of a component. Thus, method call sequence specifications specify properties of the control flow of the global execution trace. The methods in these sequences denote the use of services of other components or even of the component itself. In particular we do not exclude call-backs.

When deployed, we assume that each component contains the description of its protocol that can be checked for conformance at run-time. Only the component interfaces need to be known to the system developer, possibly with JMSeq annotations when the source code is not available. Of course it would be advantageous if the developers have documentation describing the internal components annotations. The advantage of this approach is an easy integration of JMSeq with the testing framework JUnit [5] for the execution of an individual component in a specific context to see whether they generate the expected results. In this case, we do not need the component to be annotated at all as one can easily write a JUnit test for it, annotate JUnit methods with JMSeq annotations and run the JUnit test with JMSeq.

Let us consider the notions of "black-box" and "white-box" testing. As white-box testing usually utilizes the internal structure of a software system, the source code dependence is inevitable. However, in the case of black-box testing, there is no need for the source code of the system as the test depends on the process, input and output, and the test specification of the system.

Moreover, in the field of testing, there are times that some parts of the system are not available (or made unavailable); thus, there should be a mechanism to substitute "mock implementation" for a system interface when needed. This technique may be referred to as *unit testing* in contrast with *integration testing* as different components are being regarded as stand-alone entities providing predefined interface and behavior and no mock implementation can be substituted or provided in the system. In other words, in integration testing, the system may be complete and operational on its own and in unit testing, there are points that require mock implementations (since the development is not complete yet).

JMSeq is uses black-box testing as it does not use the internal structure of the program. And, it supports both unit testing and integration testing techniques; provided that the developer or the tester provides the mock implementations required in unit testing.

Table 1 characterizes several testing frameworks by distinguishing among code annotation, code instrumentation, unit testing, and integration testing techniques.

Table 1. Approach Comparison

	Instrumentation	Annotation	Technique
JML	✓	×	Unit, Integration
JavaMOP	✓	×	Unit, Integration
PQL	✓	×	Unit, Integration
JMSeq	×	✓	Unit, Integration

3 Method Sequence Call Specification

We consider a component to be a collection of compiled Java classes. The relevant dynamic behavior of a component can be expressed in terms of specific sequences of messages between a small, fixed number of objects. In Figure 1, we see two UML message sequence diagrams each describing how the four objects interact with other and in which order. In this section, we develop a specification language for describing kinds of interactions using a context free grammar.

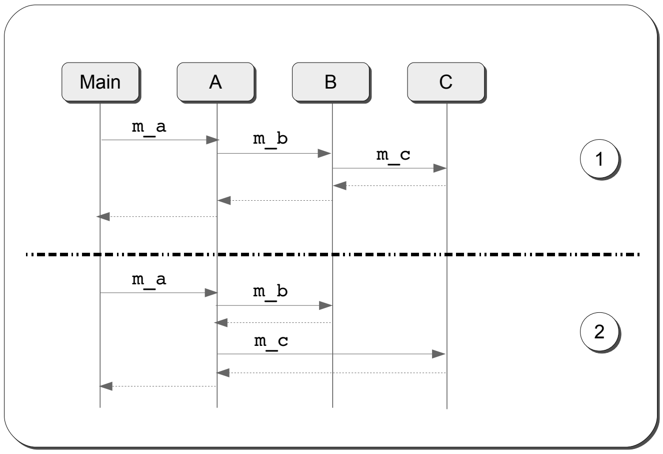


Fig. 1. Examples of Method Sequence Call Specification

Essentially, a specification language for sequences of method calls needs to distinguish between the specifications of the two cases in Figure 1.

Case 1 shows a scenario in which (the call to) m_c is nested in m_b since m_c is called during the activation of m_b (i.e. after m_b is called and before it returns). Similarly, both m_b and m_c are nested in m_a.

Case 2 represents a method call in which methods from different/same objects are called in a sequential rather than nested manner. For instance, both m_b and m_c are called by m_a.

Typically, a program will need a combination of both cases to specify its dynamic behavior. Specifying only the order of the method calls is not enough, as both cases above have the same order of method calls. It is thus required to have a specification technique that distinguishes between the *method calls* and *method returns*.

It is clear from the above examples that regular expressions over method calls are not enough to specify the typical nested call structure of a sequence of messages in the presence of recursion. In general, such sequences form a context free language. However they have a special structure: there is a deterministic pushdown automaton accepting them such that it pushes or pops at most one symbol only, depending if a method call or return is read, respectively. Such an automaton is called a visibly pushdown automaton [8].

In JMSeq, a specification denotes a post-condition associated with a method. It specifies the set of possible sequences of method calls, or protocol, of an object in the context of the relevant part of its environment. Formally, sequences of method calls belong to a context free language specified by means of a grammar. Figure 2 represents the method sequence call specification grammar.

A *specification* consists of a sequence of calls that can be repeated an a priori fixed number of times ("$\langle \mathit{Specification} \rangle^m$"), with $m \geq 0$, one or more

$$\begin{aligned}
\langle \textit{Specification}\rangle &::= \langle \textit{Call}\rangle \mid \langle \textit{Call}\rangle\ \langle \textit{Specification}\rangle \mid \\
&\quad\ \langle \textit{Specification}\rangle^{\mathrm{m}} \mid \langle \textit{Specification}\rangle\$ \mid \langle \textit{Specification}\rangle\# \mid \\
&\quad\ (\langle \textit{Specification}\rangle) \\
\langle \textit{Call}\rangle &::= \{\texttt{call}(\langle \textit{Signature}\rangle)\langle \textit{InnerCall}\rangle\}\$ \mid \\
&\quad\ \{\texttt{call}(\langle \textit{Signature}\rangle)\langle \textit{InnerCall}\rangle\}\# \mid \\
&\quad\ \{\texttt{call}(\langle \textit{Signature}\rangle)\langle \textit{InnerCall}\rangle\}^{\mathrm{m}} \mid \\
&\quad\ \{\texttt{call}(\langle \textit{Signature}\rangle)\langle \textit{InnerCall}\rangle\} \mid \\
&\quad\ \{\texttt{call}(*)\} \mid \\
&\quad\ <\ \langle \textit{Call}\rangle\ ?\ \langle \textit{Call}\rangle\ > \\
\langle \textit{InnerCall}\rangle &::= [\langle \textit{Call}\rangle]\langle \textit{InnerCall}\rangle \mid \epsilon \\
\langle \textit{Signature}\rangle &::= \langle \textit{AspectJ Call Expression Signature}\rangle
\end{aligned}$$

Fig. 2. Method Sequence Specification Grammar

times ("$\langle$*Specification*$\rangle$\$"), or zero or more times ("$\langle$*Specification*$\rangle$#"). Although JMSeq does not use aspect-oriented programming in its implementation; we have used the generic method call join points syntax of AspectJ [19], using for example # to denote the more standard Kleene star operation. Additionally, ($\langle$*Specification*$\rangle$) is a way to group specifications to avoid ambiguity. To improve readability of the specifications, grouping is only used when necessary.

A *call* is a call signature followed by a (possibly empty) sequence of inner calls. Each call can be repeated either one or more time, zero or more times, or exactly m-times, for some $m \geq 0$. Additionally, the wild card {**`call`**(∗)} denotes a call to an arbitrary method. To support branching, JMSeq also provides $<$ $\langle$*Call*$\rangle$?$\langle$*Call*$\rangle$ $>$ to allow the specification of a choice in the sequence of method executions.

Inner calls are calls that are executed before the outer call returns, i.e. they are nested. We do not have explicit return messages, but rather we specify the scope of a call by using parentheses. Information about the message call, like caller, callee, method name, and actual parameters are expressed using the generic AspectJ syntax for pointcut model that is used to express the point of calling a method from another object [19,1]. Put it simply, a call signature is of the form

```
call([ModifiersPattern] TypePattern
        [TypePattern . ] IdPattern (TypePattern | ".." , ... )
        [ throws ThrowsPattern ]
)
```

reflecting the method declarations in Java that include method names, method parameters, return types, modifiers like "static" or "public", and throws clauses. It is noticeable that the annotation are used for the "public" method from outside the component; yet, the developer is allowed other modifiers such as "private" for runtime checking since the internally used annotations are not visible from outside the component. Here `IdPattern` is a pattern specifying the method name. It can possibly be prefixed at the left by a specification of the type of the object called. Method modifiers and return type are specified by the two leftmost patterns. The method name is followed by a specification of the type of the parameters passed during the call, and possibly by a specification of the throws clause. It implies that JMSeq specification grammar can distinguish the overloaded methods in a class. Patterns may contain wild card symbols "*" and "..", used for accepting any value and any number of values, respectively. For example, the call

```
call(* *.A.m_a(..))
```

is denoting a call to method `m_a` of any object of type `A` that is placed in any package, and returning a value of any type. If there is a need to be more specific, a possible restatement of the same specification could be:

```
call(int nl.liacs.jmseq.*.A.m_a(Object, double))
```

Next we give few example of correct method sequence specifications. For instance, the specification of the sequence in case 1 of Figure 1 is given by:

```
{call(* *.A.m_a(..))[{call(* *.B.m_b(..))[{call(* *.C.m_c(..))}]}]}
```

Here is important to notice that the call to method `m_b` is internal to `m_a`, and the one to `m_c` is internal to `m_b`. The sequence in case 2 of the same figure would be:

```
{call(* *.A.m_a(..))[{call(* *.B.m_b(..))}][{call(* *.C.m_c(..))}]}
```

where both calls to methods `m_b` and `m_c` are internal to `m_a`.

These two cases depict a fixed sequence of method calls. More interesting are the cases when, for instance, the method `m_a` should be called at least once before any possible method call to `m_b` or `m_c`:

```
{call(* *.A.m_a(..))}$<{call(* *.B.m_b(..))}#?{call(* *.C.m_c(..))}#>
```

Such a specification is used in circumstances where `m_a` will satisfy a requirement in advance that is used by `m_b` or `m_c`.

It is *notable* that the meta-grammar provided in Figure 2 is a context-free grammar; however, the actual specifications are regular expressions; for example, they may contain unbounded repetition and choice of calls. They are not context free as the bound m in the repetition is assumed to be fixed and not a free variable.

4 Annotations with Method Sequence Calls

Annotations do not directly affect program semantics, but they do affect the way programs are treated by tools and libraries, which can in turn affect the behavior of the running program [3]. Annotations can be read from source files, class files, or reflectively at run-time. Once an annotation type is defined, it can be used to annotate declarations. An annotation is a special kind of modifier, and can be used anywhere that other modifiers (such as `public`, `static`, or `final`) can be used.

One of the major desired effects of using annotations in code is that it will allow for testing components without the need to have their source code. We only used the meta data loaded from the annotations during runtime.

JMSeq defines two type of annotations: sequenced object annotations and sequenced method annotations.

Sequenced Object Annotations. Simply put, `SequencedObject` annotation is just a marker for those classes to notify the annotation meta-data loader that the objects from the annotated class contain methods which specify a sequential execution. The code is demonstrated in Listing 1.

Listing 1. SequencedObject Annotation Declaration

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface SequencedObject {
    // we need no properties for this annotation as this is only a marker.
}
```

`@Retention(RetentionPolicy.RUNTIME)` declares that this annotation is only applicable during runtime and may not be used in other scenarios, whereas `@Target(ElementType.TYPE)` declares that this annotation can only be used on types including classes, interfaces and enumerated types.

Sequenced Method Annotation. A sequence method annotation is used to specify the sequence of method calls under a given method. The annotation requires a string property declaring the sequential specification discussed in Section 3. Listing 2 presents the declaration.

Listing 2. SequencedMethod Annotation Declaration

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface SequencedMethod {

    String value();

    Class<VerificationFailureHandler> verificationFailureHandler();
}
```

`@Target.ElementType.METHOD` declares that this annotation is only applicable to *methods*. The string value from `value()` holds the sequential specification. The class `VerificationFailureHandler` is introduced by `verificationFailureHandler()` and is used when a sequence execution failure occurs at runtime. Its implementation is left to the tester or developer who should provide a custom behavior to handle the verification failures.

In Listing 3 we give two examples of annotations of the class `Main` class in Figure 1. In both cases we annotated method `main()` with two sequences of method calls, describing the behaviors given in Figure 1.

Listing 3. Sample annotated specification

```

// Case 1
@SequencedObject
public class Main {

    @SequencedMethod("{call(* *.A.m_a(..))[{call(* *.B.m_b(..))[{call(* *.C.
        m_c(..))}]}]}")
    public void main() {
        // ...
    }

    public void init() {
        // ...
    }
}

// Case 2
@SequencedObject
public class Main {

    @SequencedMethod("{call(* *.A.m_a(..))[{call(* *.B.m_b(..))}][{call(* *.C
        .m_c(..))}]}")
    public void main() {
        // ...
    }

    public void init() {
        // ...
    }
}
```

5 The JMSeq Framework

In this section, we present the JMSeq monitoring and testing framework, and discuss its implementation that uses Java 5 Annotations [3] and Java Platform Debugger API [4]. More implementation details, including samples and documentation, can be found at `http://code.google.com/p/jmseq/`.

As discussed above, we assume that Java components come together with sequenced object and sequenced method annotations. More annotations are possible within the same class, but not for the same method. This implies a local and partial view of the component specification, that can be scattered among all its constituent classes. The annotated methods are the ones that will be monitored by JMSeq.

JMSeq, based on some initial parameters, initiates another *inner Java virtual machine (JVM)* inside the current execution to control the sequenced execution of the program (Figure 3). Essentially, the parameters tell the inner JVM what type of events are going to be reported back to JMSeq for verification such as method entries and method exits. Also, JMSeq is not interested in events from all objects but only for those specified in the parameters. The inner JVM takes advantage of the Java Platform Debugger Architecture (JPDA) to access the needed details on the execution of the system while it is running. JPDA is a framework for debugging and interfacing the JVM. Java Debugger Interface is the interface of JPDA that gives access to different details on the execution of a program.

Before execution, JMSeq inspects the classes in the running class path to collect and store all the methods that are annotated for sequenced verification. This step, denoted in Figure 3, does not need code inspection, but uses the annotated meta data available in the compiled code.

The program is now executed. Whenever an event is reported to JMSeq the event trace model is updated in such a way that events are aware of their previous call stack trace (Figure 3): if the event represents a method that does not need to be monitored the execution continues until the next event, otherwise it is verified if it is an expected event. The verification is done through a simple state machine with a stack for the nested events occurred so far. The verification process is described as (Figure 3):

1. A “call expression” of the event model is constructed.
2. Using the meta data available from annotations of the methods, then, the next “possible call expressions” of the current state is built.
3. A match making is done between the possible call expressions and the current call expression as the candidate. The possible call expressions are computed on-the-fly rather than constructing the full automaton in the beginning of the verification process. This way we avoid the construction of states in the automaton that are never used in the execution. To avoid repeated computations of the same states, JMSeq utilizes dynamic programming techniques. If a match is found, the method is accepted; otherwise it fails. When a failure occurs, JMSeq will execute the custom verification handler that is implemented either internally as part of the component code by the programmer, or externally by the system designer.

The overall JMSeq process is depicted in Figure 3.

5.1 JMSeq Architecture

The overall architecture of JMSeq is given in Figure 4. It basically consists of three main modules: one for handling the communication with the JVM executing the program, another module for storing the annotation information, and a third module for executing the run-time verification.

The current design of JMSeq is completely *general* and MODULAR; as it allows for replacing the grammar in Figure 2 with other specification modules, based, for example, on temporal logics or extended regular expressions.

Program Execution Trace Model and Processing. According to JDI event model, JMSeq takes control over some of the execution events that JVM publishes during a program execution. Therefore, a component was designed to

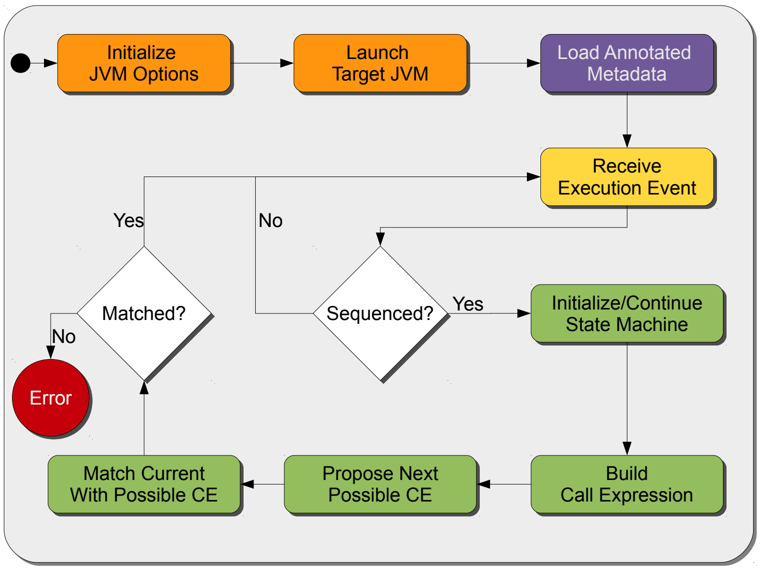


Fig. 3. Runtime Object Monitoring and Sequenced Execution Verification

model and hold the execution trace events required for event handling and execution verification.

1. *Execution* is the central component that holds the information about every execution in the JVM using the JDI event mechanisms. Relevant events include "method entry" and "method exit". It provides access to information such as:
   – The object that is currently executing (*the callee object*) and its unique identifier in JVM.
   – Event details through subclasses such as method return values or *the caller object* reference in case of a method exit event.
   – *Parent Execution*: every execution can hold a reference to its parent execution object forming a directed tree of executions. This help traversing the executions at validation time or for the simulation of formal specification.
2. *EventHandler* is the event handling interface that is injected into JVM with access to JDI information. The event handler receives event for which it is subscribed and possibly takes an associate action. In particular it creates an instance of `Execution` for each sequence annotation and it stores it in the `ExecutionTraceOracle` registry ready to be used by the verification module. In general, all events received and processed by `EventHandler` are stored in this registry for further use by other components.

Sequential Execution Annotation Repository. As the execution traces are stored in a repository, they are supposed to be checked and verified

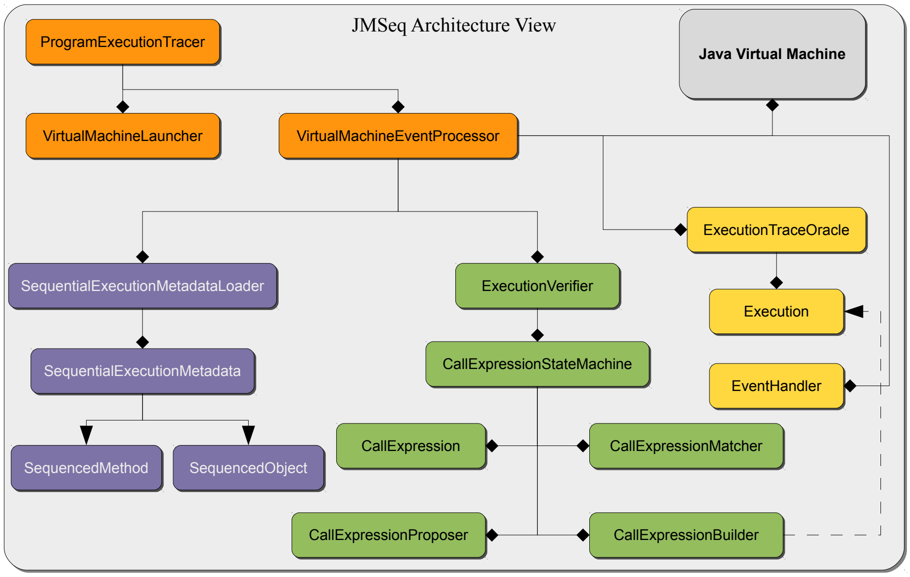


Fig. 4. Software Architecture for Method Sequence Specification

against the formal specifications as described on the `@SequencedObject` and `@SequencedMethod` annotations of the compiled classes in the program. It is the task of sequential execution annotation repository to store this information. A service is responsible to read and load the meta-data of all classes that are annotated. This information is used when there is a need to verify the conformance of an execution event.

Execution Verification. During an execution, events are received that need to be monitored and verified against a message sequence specification. For every specification in the meta-data repository, a deterministic state machine *including a stack* is created for recognizing the sequences in the specification. Whenever an event is processed that belongs to a sequence of method calls, the execution verifier checks if the state machine can execute a transition associated with this event. If the state machine accepts the current event, the execution will continue; otherwise, the execution is "invalid" and therefore it is stopped. At this point, JMSeq provides a simple way to plug in a verification failure handler by means of a class provided by either the component designer or the test developer. The verification failure handler should implement an interface introduced by JMSeq.

The execution verification component is composed by the following elements:

1. *Call Expression* is a simple component for interacting with each state machine object. Basically, it transforms execution events coming from the JVM into call expressions used by the state machine components.

2. *Call Expression State Machine*: Sequences of executions are translated to "call expression"s that are to be accepted by a state machine. The stack of the automaton is necessary to distinguish the method's context on different calls to the same method calls, for example. At the end of a successful sequential execution, the stack for the automaton associated to that sequenced execution specification should be empty. Otherwise the top of the stack is used to match the event with a possible candidate call expression of the previous event.
3. *Call Expression Builder* constructs a call expression out of:
   (a) A string which is in the format of the JMSeq specification grammar as in Figure 2. This service is used the first time a sequential execution is detected to build the root of the future possible (candidate) call expressions.
   (b) An execution event; every time an execution is handed over to verification module, an equivalent call expression is built for it so that it can be compared and matched against the call expression for the previous event.
4. *Call Expression Proposer* is a service proposing *possible next call expressions* for a given call expression based on the sequences specified in the annotation. As described by the grammar in Figure 2, for each current call expression there can be several possible next call expressions that may appear as the next event, but for each of them the automaton associated with the grammar can only make one transition (that is, the specification is a deterministic context free language). Note that since more specification sequences are possible involving the same method, only those call expressions that are valid in all specifications will be proposed.
5. *Call Expression Matcher* is another service that tries to match two call expressions. It is used, for example, to validate the current call expression against all those proposed by the previous service. If a match is found, the execution continues; otherwise the verification is regarded as failed. In this case, if a verification failure hander is provided, the failure data is transferred to it for further processing.

6 Related Work

JML [22] provides a robust and rigorous grounds for specification of behavioral checks on methods. Although JML covers a wide range of concerns in assertion checking, it does *not* directly address the problem of method sequence call specification as it is more directed towards the reasoning about the state of an object. JML is rather a comprehensive modeling language that provides many improvements to other extensions to Design by Contract (DBC) [24] equivalent formalism such as jContractor [9] and Jass [11].

In [16], taking advantage of the concept of *protocols*, an extension to JML is proposed that provides a syntax for method sequence calls (protocols) along with JML's functional specification features. Through this extension, the developer can specify the methods' call sequence through a *call sequence clause* in JML-style meta-code. In the proposed method, the state of a program is modeled as a "history" of method calls and return calls using the expressiveness of regular expressions; thus, a program execution is a set of "transitions" on method call histories. The verification takes place when the execution history is simulated using a finite state machine and checked upon the specified method call sequence clause.

JML features have been equivalently implemented with AspectJ constructs [26], using aspect-oriented programming [19] They propose AJMLC (AspectJ JMLC) that integrates AJC and JMLC into one compiler so that instead of JML-style meta-code specifications, the developer writes *aspects* to specify the requirements.

In [17] an elegant extension of JML with histories is presented. Attribute Grammars [21] are used as a formal modeling language, where the context-free grammar describes properties of the control-flow and the attributes describe properties of the data-flow, providing a powerful separation of concerns. A run-time assertion checker is implemented. Our approach differs in several respects. The implementation of their run-time checker is based on code instrumentation. Additionally, they use local histories of objects, so callbacks can not be modeled. However, the behavior of a stack can be modeled in their approach (and not in ours), since their specifications are not regular expressions but context-free languages.

In the domain of runtime verification, Tracematches [7] enables the programmer to specify events in the execution trace of a program that could be specified with "the expressiveness" of a regular pattern. The specification is done with AspectJ pointcuts and upon a match the advised code is run for the pointcut. Along the same line, J-LO [12] is a tool that provides temporal assertions in runtime-checking. J-LO shares similar principles as Tracematches with differences in specifications using linear time temporal logic syntax.

Additionally, Martin, Livshits and Lam propose PQL [23] as a program execution trace query language. It enables the programmer to express queries on the execution events of objects, methods and their parameters. PQL then takes advantage of two "static" and "dynamic" checkers to analyze the application. The dynamic checker instruments the original code to install points of "recovery" and "verification" actions. The dynamic checker also translates the queries into state machine for matching criteria. The set of events PQL can deal with includes method calls and returns, object creations and end of program among others. Accordingly, generic logic-based runtime verification frameworks are proposed as in MaC [20], Eagle [10], and PaX (PathExplorer) [18] in which monitors are instrumented using the specification based on the language specific implementations.

Using runtime verification concepts, Chen and Rosu propose MOP [15,6] as a generic runtime framework to verify programs based on *monitoring-oriented programming*. As an implementation of MOP, JavaMOP [14] provides a platform supporting a large part of runtime JML features. Safety properties of a program are specified and inserted into the program with monitors for runtime verification. Basically, the runtime monitoring process in MOP is divided into two orthogonal mechanisms: "observation" and "verification". The former stores the desired events specified in the program and the latter handles the actions that are registered for the extracted events. Our approach follows the same idea as MOP, but it does not use AOP to implement it. Another major difference is in the specifications part. MOP specifications are generic in four orthogonal segment: logic, scope, running mode and event handlers. Very briefly, the *scope* section is the fundamental one that defines and specifies the parts of the program under test. It also enables the user to define desired events of the program that need to be verified. The *logic* section helps the user specify the behavioral specification of the events using different notations such as regular expressions or context-free grammars. The *running mode* part lets the user specify what is the running context of the program under test; for instance, if the test needs to be run per thread or in a synchronized way. And, *the event handlers* section is the one to inject customized code of verification or logic when there is a match or fail based on the event expression logic.

In Listing 4 we show an example of JavaMOP code with ERE logic for the two specifications given in Figure 1.

Listing 4. Sample JavaMOP Specification using CFG logic

```
SampleCase1(Main m) {
    event method_m_a before(A a):
        call(* A.m_a(..)) && target(a) {}
    event method_m_b before(B b):
        call(* B.m_b(..)) && target(b) && cflow(SampleCase1_method_m_a) {}
    event method_m_c before(C c):
        call(* C.m_c(..)) && target(c) && cflow(SampleCase1_method_m_a) &&
             cflow(SampleCase1_method_m_b) {}

    ere: (method_m_a method_m_b method_m_c)*

    @fail {
        System.err.println("Invalid Execution");
        __RESET;
    }
}

// Case 2
SampleCase2(Main m) {
    event method_m_a before(A a):
        call(* A.m_a(..)) && target(a) {}
    event method_m_b before(B b):
        call(* B.m_b(..)) && target(b) && cflow(SampleCase1_method_m_a) {}
    event method_m_c before(C c):
        call(* C.m_c(..)) && target(c) && cflow(SampleCase1_method_m_a) && !
             cflow(SampleCase1_method_m_b) {}

    ere: (method_m_a method_m_b method_m_c)*

    @fail {
        System.err.println("Invalid Execution");
        __RESET;
    }
}
```

It is interesting to note that both MOP specifications have *the same ERE expression*. This is because JavaMOP has separated logic and scope specification from event handling. The difference in monitoring is obtained by the different usage of the command `cflow` in the scope section. This command is an AspectJ construct to control the context of the execution when running some code inside a method [19].

7 Conclusion and Future Work

We proposed JMSeq, a framework for specifying sequences of method calls using Java annotations. The sequences do not only consist of method names, but may contain information such as object caller and callee. JMSeq uses Java Platform Debugger to monitor the execution of a component based system based on Java. Monitoring is divided into two phases: observing the events in the program and verifying them against the local specifications provided through annotated objects at runtime. No code instrumentation is necessary, as only binary code is enough for system testing and monitoring purpose. JMSeq can be integrated with JUnit for unit testing purposes. An initial version of JMSeq runtime checker is available at `http://code.google.com/p/jmseq`.

We believe JMSeq is a novel approach to runtime verification of software using code annotations. The approach is especially suitable for runtime component based verification, as it does not require the presence of source code. In line with this end, we plan to extend the support of JMSeq by providing native features such as mock implementation or symbolic execution in case parts of the system are unavailable which is particularly useful in unit testing. Currently, JMSeq only provides testing capabilities in the context of JUnit.

Another potential area of improvement is the data model used by JMSeq. Currently it overlaps with the event model that JPDA provides when publishing the registered events in JVM. The resulting overhead is rather expensive. Optimization will allow, for example, to store only the minimal necessary information providing a faster indexing for the retrieval of the events. Further performance improvement can be obtained by better exploiting the connections between JPDA and JVM. For example, there are tools such as Eclipse Debug Platform [2] that provide extensive facilities through JPDA with high performance. Currently, in JMSeq, it is the standard JVM that runs the program and publishes the events that are interesting to JMSeq for further verification for which, in turn, JMSeq uses a simple state machine to verify the executing events. As another future work, instead of using a state machine, one could use a pushdown automaton which allows for context-free method call sequence specifications. Moreover, in another approach, JVM, JPDA and the state machine can be merged together such that it is actually the JVM that asks permission for the next execution from the state machine that is provided. Thus, the testing framework can even take control of the underlying program execution for further checks or verification. In other words, method call sequence specifications may also be used in the pre-conditions of a method. In comparison with the runtime checking, another line of future work can be to extend JMSeq to support static verification of protocols.

Acknowledgments. We thank the referees for their comments and Michiel Helvensteijn for the helpful discussion.

References

1. AspectJ Language Semantics, `http://eclipse.org/aspectj/doc/released/progguide/semantics-pointcuts.html`
2. Eclipse Debug Platform, `http://www.eclipse.org/eclipse/debug/`
3. Java 5 Annotations, `http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html`
4. JPDA Reference Home Page, `http://java.sun.com/javase/technologies/core/toolsapis/jpda/`
5. JUnit Test Framework, `http://www.junit.org/`
6. MOP: Monitoring-oriented programming, `http://fsl.cs.uiuc.edu/index.php/MOP`
7. Allan, C., Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhotak, O., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: Adding trace matching with free variables to AspectJ. In: OOPSLA (2005)
8. Alur, R., Madhusudan, P.: Adding nesting structure to words. J. ACM 56, 16:1–16:43 (2009)
9. Anercrombie, P., Karaorman, M.: jContractor: Bytecode instrumentation techniques for implementing dbc in Java. In: RV 2002 (2002)
10. Barringer, H., Goldberg, A., Havelund, K., Sen, K.: Rule-Based Runtime Verification. In: VMCI 2004 (2004)
11. Bartetzko, D., Fischer, C., Moller, M., Wehrheim, H.: Jass - Java with Assertions. In: RV 2001 (2001)
12. Bodden, E.: J-lo, a tool for runtime-checking temporal assertions. Master Thesis, RWTH Aachen University (2005)
13. Chen, F., Rosu, G.: Towards Monitoring-Oriented programming: A paradigm combining specification and implementation. Electronic Notes in Theoretical Computer Science 89(2), 108–127 (2003)
14. Chen, F., Rosu, G.: Java-MOP: A monitoting oriented programming environment for Java. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 546–550. Springer, Heidelberg (2005)
15. Chen, F., Rosu, G.: MOP: An Efficient and Generic Runtime Verification Framework. In: OOPSLA. ACM Press, New York (2007)
16. Cheon, Y., Perummandla, A.: Specifying and checking method call sequences of Java programs. Software Qual. J. 15, 7–25 (2007)
17. de Gouw, S., Vinju, J., de Boer, F.S.: Prototyping a tool environment for run-time assertion checking in JML with Communication Histories. In: FTfJP 2010 (2010)
18. Havelund, K., Rosu, G.: Monitoring Java programs with Java PathExplorer. In: RV 2001 (2001)
19. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.: Getting started with ASPECTJ. In: ACM CACM (2001)
20. Kim, M., Kannan, S., Lee, I., Sokolsky, O.: Java-MaC: a Runtime Assurance Tool for Java. In: RV 2001 (2001)

21. Knuth, D.E.: Semantics of context-free languages. Mathematical Systems Theory 2(2), 127–145 (1968)
22. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary Design of JML: A Behavioral Interface Specification Language for Java. ACM SIGSOFT Software Engineering (2006)
23. Martin, M., Livshits, V.B., Lam, M.S.: Finding application erros and security flaws using PQL: a program query language. In: OOPSLA (2005)
24. Meyer, B.: Object-Oriented Software Construction, 2nd edn. Prentice Hall, New Jersey (2000)
25. Necula, G.C.: Proof-carrying code. In: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1997, pp. 106–119. ACM, New York (1997)
26. Rebelo, H., Soares, S., Lima, R., Borba, P., Cornelio, M.: JML and Aspects: The benefits of instrumenting JML features with AspectJ (2008)
27. Szyperski, C., Gruntz, D., Murer, S.: Component software: beyond object-oriented programming. Addison-Wesley, Reading (2002)

An Introduction to Pervasive Interface Automata[⋆]

M. Calder[⋆⋆], P. Gray, A. Miller, and C. Unsworth

Computing Science, University of Glasgow, U.K.
`Muffy.Calder@glasgow.ac.uk`

Abstract. Pervasive systems are often context-dependent, component based systems in which components expose interfaces and offer one or more services. These systems may evolve in unpredictable ways, often through component replacement. We present pervasive interface automata as a formalism for modelling components and their composition. Pervasive interface automata are based on the interface automata of Henzinger et al [3], with several significant differences. We expand their notion of input and output actions to combinations of input, output actions, and callable methods and method calls. Whereas interface automata have a refinement relation, we argue the crucial relation in pervasive systems is component *replacement*, which must include consideration of the services offered by a component and assumptions about the environment. We illustrate pervasive interface automata and component replacement with a small case study of a pervasive application for sports predictions.

1 Introduction

Pervasive systems are often context-dependent, component based systems that can evolve in unpredictable ways, through component addition (composition) and replacement. But unpredictability can have detrimental consequences for usability and for wide-spread adoption of systems: *how can we make component based evolution more predictable*?

The question is difficult because components are often designed and implemented incrementally by different development teams, or via end-user configurations, or they are mashups (e.g. make your own Android application mashup[7]). A traditional approach involving modelling and reasoning about full behavioural specifications is unlikely to be plausible. Consequently, we focus on the *interfaces* exposed by components and the *services* they offer. We define *pervasive interface automata* as a formalism for modelling the interfaces exposed by components and their composition. These automata are based on the interface automata of Henzinger et al [3], but there are several significant differences.

[⋆] This work was funded by the EPSRC grant, *Verifying Interoperability Requirements in Pervasive Systems* EP/F033206/1.

[⋆⋆] Corresponding author.

First, we expand their notion of input and output actions to combinations of input/output and calling and callable methods. We refer to the latter two as master and slave actions, respectively. When components are composed, they synchronise on shared actions so that input/output and calling/called behaviour assumptions are met. Informally, this means that input/output and master/slave behaviours are (two-way) synchronised. We relax the synchronisation of components to allow a component offering a master action to wait until the appropriate slave action is offered. This allows both *busy send* and *busy receive*, where the original interface automata only allows a component to wait to receive, and not wait to send.

Second, we argue the crucial relation in pervasive systems is component *replacement*, which must include consideration of both services offered by a component and assumptions about the environment, where the environment is a composition of components. We include the environment because it may include actions that affect the interface of the component under consideration; specifically, it can cause some actions to become hidden (internal) or some choices to be removed.

As an example, consider a server component within a sports prediction application. The application keeps track of fixtures and results (e.g. a football league, or a tennis tournament) and allows users to make and share predictions in advance of actual events. The standard server component offers a service to *add predictions* and to *get predictions.* An online betting company might offer an alternative component that offers a service to *place a bet*, in addition to the previously mentioned services, the delivery of which relies on the availability of services *get data* and *add data* supplied by the betting company's online server component. Under what circumstances can we replace the standard component by the betting company's component? The latter relies on services from the company's online server component, which we call *environmental assumptions*; informally, we will allow replacement when environmental assumptions are met.

We introduce a linear temporal action logic to define service behaviour and define the satisfaction of the formulae by a pervasive interface automaton under assumptions about the environment. For a given environment, we can replace one component by another, with respect to a service, when both components satisfy the service in the environment. We judge the quality of a replacement of one component by another by the number of services that it preserves and any new services it may offer. We illustrate pervasive interface automata and component replacement with a small case study of an application for sports predictions, based on a real application.

In summary, the contributions of the paper are the following:

- definition of pervasive interface automata
- definition of action matching and algorithm for composition of pervasive interface automata
- logical specification of services
- satisfaction relation of pervasive interaction automaton and a service under environmental assumptions

– replacement relation between components, services and environment assumptions
– application of pervasive interaction automata and replacement relation to a case study involving sports predictions.

In the next section we give a brief overview of our case study, as motivation for pervasive interface automata, which we define formally in Sect. 3. In Sect. 4 we discuss action matching and we define automata composition by way of an algorithm. In Sect. 5 we introduce the concept of a service and define an action based logic for defining service behaviour; in the following section, Sect. 6 we define the replacement of one component for another, with respect to an environment and a set of services. Comparison of pervasive interface automata with interface automata and other related work is in Sect. 7. Our conclusions and directions for future work are in Sect. 8.

2 Case Study

We now present a case study which we use both as motivation and for explanation. The case study is a pervasive application for sports prediction, written within the Domino framework [2]. This application is mobile phone based and allows a user to see a list of upcoming sports fixtures and make predictions about the results. These predictions are then uploaded and stored on a remote server. A prediction league graphical interface allows a user to download predictions from multiple users (stored on a remote sever component) to compare them with the actual results in a league table. The architecture of this system is shown in Fig 1. Brief descriptions of the components follow.

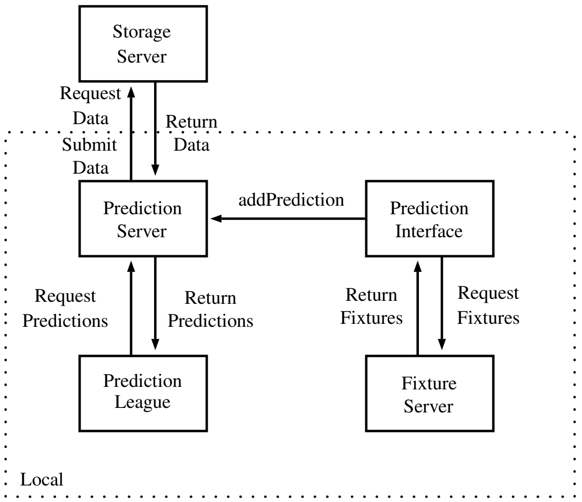


Fig. 1. Architecture of the sports prediction application

Prediction Interface. This component is a graphical user interface that allows a user to pull information about upcoming sporting fixtures from a fixture server. A user can input a prediction for one or more of the fixtures. The predictions are then sent to a prediction server for storage.

Fixture Server. This is a passive component that responds to requests for fixture information. The data returned is a list of forthcoming sporting events, including information about times, dates, locations and the teams involved in the fixture.

Prediction Server. This component accepts predictions from a prediction interface component and stores them in an external storage server component. The information sent to the server includes the fixture, the prediction and the user that made the prediction. Predictions are also retrieved from the storage server upon request.

Storage Server. This is a web based generic data storage server component. It allows the storage of data and allows any user to retrieve any stored information.

Prediction League. This component is a graphical user interface that allows a user to pull historical predictions from a prediction server along with the actual results to evaluate the predictions. The prediction league component can also retrieve and display the predictions from other users, allowing a user to compare their performance against that of their friends.

In this system most components are held on a local device (such as a mobile phone) only the storage server component is external. A user may wish to upgrade or replace any of the individual components to increase the overall systems functionality or to improve the user experience. However, any new component must be capable of providing all the services provided by the component it is replacing.

3 Pervasive Interface Automata

Pervasive Interface Automata are an extension of interface automata [3] with pervasive systems in mind. The main difference is the addition of annotations to actions. These annotations include ! and ? to indicate output and input respectively, and $\circ$ and $\star$ to represent slave and master actions respectively. Master actions are instigated by the component and slave actions are instigated by the environment (i.e. some other component). We have distinguished master/slave behaviour from input/output behaviour to ensure that we capture the notion of when a component requires external resources to function, i.e. to deliver a service. This is essentially the difference between *calling* a method and offering a method that *can be called*. For example, consider the four combinations of behaviour. If automaton P offers action $foo?^{\circ}$, then P is offering a callable method foo, which will receive data; if P offers action $foo?^{\star}$, then data is returned to P, from a method instigated by P. If automaton P offers action $foo!^{\circ}$, then P

is offering a callable method foo, which delivers data; if P offers action $foo!^\star$, then data is sent by P, from a method instigated by P.

Masters synchronise with slaves, and inputs synchronise with outputs. However, there is an asymmetry between masters and slaves. Whereas components with master actions (i.e. method calls) *require* slave actions (i.e. callable methods) in order to function properly, the converse is not true. More precisely, if a component reaches a state in which a master action is offered, a synchronising slave action is required at that point. On the other hand, if a component reaches a state in which a slave action is offered, and there is no corresponding synchronising master action, that slave action can be considered spare capacity; that is the action is on offer, but no other component requires it. In these circumstances, it can be ignored.

Definition 1. *A Pervasive Interface Automaton* $P = \langle V_P, V_P^{init}, A_P, T_P \rangle$ *where*

- $V_P = \{v_1, v_2, \ldots, v_{|V_P|}\}$ *is a finite set of states*
- $V_P^{init} \subseteq V_P$ *is the set of initial states*
- $A_P = \{a_1, a_2, \ldots, a_{|A_P|}\}$ *is the finite set of actions,*
  - *where an action* $a = name[?^\star|?^\circ|!^\star|!^\circ]$
- $T_P \subseteq V_P \times A_P \times V_P$ *is the set of steps (state transitions)*

Action annotations indicate the following. Annotations ! and ? denote input and output, respectively, and $\star$ and $\circ$ denote master and slave, respectively. An action that has no annotation is a hidden (internal) action. A_P° is defined to be the set of all slave actions, i.e. actions with a $\circ$ annotation, and $A_P^\star$ is defined to be the set of all master actions. We often use graphical representations of example automata.

To aid later description, a number of functions are now defined.

- $A_P(v)$ is the set of actions enabled at state v
  - an action a is enabled in state v if there exists $(v, a, v') \in T_P$
  - $A_P^\circ(v)$ is the set of slave actions enabled at state v
  - $A_P^\star(v)$ is the set of master actions enabled at state v
- $source(t)$ returns the state v, where transition $t = (v, a, v')$
- $act(t)$ returns the action a, where transition $t = (v, a, v')$
- $target(t)$ returns the state v', where transition $t = (v, a, v')$
- $\rho = \{t_1, t_2, \ldots\}$ is a path, defined as an ordered multiset of transitions
  - ρ_i the i^{th} transition in ρ
  - $\forall t_i, t_{i+1} \in \rho, target(t_i) = source(t_{i+1})$
  - $\rho \in P$ means that $\forall \rho_i \in \rho \implies \rho_i \in T_P$

An automaton P is said to be closed if it does not require any external resources to function. Input/output actions require external resources to function. However, slave actions (annotated with $\circ$) are assumed to be spare capacity, thus they will not be performed unless synchronised with some other component. Therefore, only master actions need be considered when determining if an automaton is open or closed. However, it may be the case that some master actions are only enabled in states that require a transition involving a slave action to be taken to be reached from the initial state. Such unreachable actions will not be considered. The set of input/output requirements for an automaton is now defined.

Definition 2. *The set of input/output requirements for automaton P is*

$$req(P) = \{a \in A_P^{\star} | \exists \rho \in P, \exists t_i \in \rho, act(t_i) = a, \forall j < i, act(t_j) \notin A_P^{\circ}\}$$

If $req(P) = \emptyset$ then P is closed and P is open otherwise.

4 Composition

Before two automata can be composed, it needs to be established how they are to interact. This is a non-trivial problem, which will need a domain specific solution. For the purposes of this document, it will be assumed that automata will synchronise on action names. For example, if automaton P has an action $foo?^{\circ}$ and automaton Q has the action $foo!^{\star}$, the composition $P \otimes Q$ will have both actions combined into a single hidden action *foo*. The set $match(P, P \otimes Q)$ is used to show how actions in the automaton P are mapped to the actions in the product automaton $P \otimes Q$. For example in the case mentioned above, $(foo?^{\circ}, foo) \in match(P, P \otimes Q)$, meaning the slave input action *foo* in P is mapped to the hidden action *foo* in $P \otimes Q$. Similarly, $(foo?^{\star}, foo) \in match(Q, P \otimes Q)$. For each action $a \in A_P$ we assume there is a corresponding pair $(a, a') \in match(P, P \otimes Q)$, such that neither a nor a' appear in any other pair in $match(P, P \otimes Q)$. In other words, a matches, or synchronises uniquely with a' in $P \otimes Q$. Note, our notion of 2-way synchronisation is similar to CCS [10], where action a matches action $\overline{a}$. But, whereas in CCS a and $\overline{a}$ synchronise to become hidden τ, we assume the name of the hidden action is retained. Note that, $match(P, P \otimes Q)$ is used to map all actions from P to $P \otimes Q$ not just the synchronised actions.

For the default case of matching actions, if $a_P \in A_P$ and $a_Q \in A_Q$ are to be synchronised in $P \otimes Q$ then a_P and a_Q must be compatible. Meaning that master output actions match slave input actions and master input actions match slave output actions. Both a_P and a_Q are matched to the same hidden action in $P \otimes Q$. Alternately, if an action $a_P \in A_P$ is not to be synchronised then it will be unchanged in $P \otimes Q$.

Pervasive interface automata P and Q are composable if each automaton can perform its shared actions as required. That is, whenever P has an enabled shared master action either Q is also capable of performing the corresponding slave action, or P is able to wait until Q is ready to perform the slave action. In practice this means that if a shared master action a is enabled in a state $v \in V_P$ but not enabled in a product state $(v, u) \in V_{P \otimes Q}$ then all paths originating in (v, u) must include a state in which a is enabled. When allowing an automaton to wait we do not distinguish between input and output actions, therefore, we allow both busy send and busy receive. Note that while P is waiting Q may need to interact with one or more other components, in which case the availability of these components will be a factor in the assessment of the composability of P and Q. Such requirements can be modelled as environmental assumptions, meaning that for P and Q to be composable the environment must meet these assumptions.

4.1 Composition

The composition of automata has two stages. The first is to generate the product of the two automata. The second attempts to validate the product as a valid composition.

We first define *shared*(P, Q) as the set of shared actions in the product of $P \otimes Q$:

$$\mathit{shared}(P, Q) := \{a | (a_P, a) \in \mathit{match}(P, P \otimes Q)\} \cap \{a | (a_Q, a) \in \mathit{match}(Q, P \otimes Q)\}$$

Product. We now define the product of automata $P \otimes Q$ as:

$$\begin{array}{l}
V^{init}_{P\otimes Q} = V^{init}_P \times V^{init}_Q \\
A_{P\otimes Q} = \{a' | (a, a') \in \mathit{match}(P, P \otimes Q)\} \cup \{a' | (a, a') \in \mathit{match}(Q, P \otimes Q)\} \\
T^{prov}_{P\otimes Q} = \\
\qquad \{((v_1, u), a', (v_2, u)) |\ u \in V_Q, ((v_1), a, (v_2)) \in T_P, \\
\qquad\qquad (a, a') \in \mathit{match}(P, P \otimes Q), a' \notin \mathit{shared}(P, Q)\} \cup \\
\qquad \{((v, u_1), a', (v, u_2)) |\ v \in V_P, ((u_1), a, (u_2)) \in T_Q, \\
\qquad\qquad (a, a') \in \mathit{match}(Q, P \otimes Q), a' \notin \mathit{shared}(P, Q)\} \cup \\
\qquad \{((v_1, u_1), a', (v_2, u_2)) |\ ((v_1), a_1, (v_2)) \in T_P, ((u_1), a_2, (u_2)) \in T_Q, \\
\qquad\qquad (a_1, a') \in \mathit{match}(P, P \otimes Q), \\
\qquad\qquad (a_2, a') \in \mathit{match}(Q, P \otimes Q)\} \\
s \in V_{P\otimes Q} \iff \exists \rho = \{\rho_1, \ldots, \rho_n, \ldots\}.\ \mathit{source}(\rho_1) \in V^{init}_{P\otimes Q} \\
\qquad\qquad \wedge \quad \mathit{target}(\rho_n) = s \\
\qquad\qquad \wedge \quad \forall \rho_i \in \rho.\ \rho_i \in T^{prov}_{P\otimes Q} \\
t \in T_{P\otimes Q} \iff t \in T^{prov}_{P\otimes Q} \wedge \mathit{source}(t) \in V_{P\otimes Q}
\end{array}$$

The set of initial states of $P \otimes Q$, $V^{init}_{P\otimes Q}$ is the product of the two sets of initial states V^{init}_P and V^{init}_Q. The action set $A_{P\otimes Q}$ is the union of the action sets of P and Q, respecting the matchings $\mathit{match}(P, P \otimes Q)$ and $\mathit{match}(Q, P \otimes Q)$ and $T^{prov}_{P\otimes Q}$ is the provisional set of transitions for $P \otimes Q$, which is used only as a construct to aid the definition of the product. A set of transitions is added to $T^{prov}_{P\otimes Q}$ for each non-shared transition in T_P; for a transition (v_1, a, v_2) this set consists of a transition $((v_1, u), a', (v_2, u))$ for each $u \in V_Q$, where $(a, a') \in \mathit{match}(P, P \otimes Q)$. Similarly, a set of transitions is added to $T^{prov}_{P\otimes Q}$ for each non-shared transition in T_Q. For every pair of transitions $t_p \in T_P$ and $t_q \in T_Q$, where t_p and t_q involve matching shared actions, a transition t is added to $T^{prov}_{P\otimes Q}$, where $\mathit{source}(t) = (\mathit{source}(t_p), \mathit{source}(t_q))$, $\mathit{target}(t) = (\mathit{target}(t_p), \mathit{target}(t_q))$ and $\mathit{act}(t) = a$, where $(\mathit{act}(t_p), a) \in \mathit{match}(P, P \otimes Q)$. The set of states $V_{P\otimes Q}$

contains all states reachable via a path constructed of transitions from $T^{prov}_{P\otimes Q}$ and originating from a state in $V^{init}_{P\otimes Q}$. Finally, the set of transitions $T_{P\otimes Q}$ consists of all transitions $t \in T^{prov}_{P\otimes Q}$, where $source(t) \in V_{P\otimes Q}$.

Composition Validation. Informally, a product is a valid composition if the following two properties hold for master transitions in P (and Q, respectively). First, for every transition in P that involves a master action, there is a corresponding transition in $P \otimes Q$. Second, if v is a state of $P \otimes Q$ at which master action a is enabled, and a corresponds to a master action of P, then there is a path prefix in $P \otimes Q$ that contains a, all actions occurring prior to a are hidden, and they do not involve a state change for P.

The product $P \otimes Q$ is a valid composition iff:

$$\begin{aligned}
&\forall t \in T_P.\ act(t) \in A^\star_P \implies \\
&\qquad \exists t' \in T_{P\otimes Q}.\ (t' = ((source(t), \_), a, (target(t), \_)) \\
&\qquad\qquad \wedge\ (act(t), a) \in match(P, P \otimes Q)) \\
&\quad \wedge\quad \forall v \in V_{P\otimes Q}.\ (v = (source(t), u)) \implies \\
&\qquad\qquad \exists path \rho = \{\rho_1, \ldots, \rho_n, \ldots\}.\ (\rho \in P \otimes Q \\
&\qquad\qquad\qquad \wedge\ source(\rho_1) = (source(t), u) \\
&\qquad\qquad\qquad \wedge\ act(\rho_n) = a \\
&\qquad\qquad\qquad \wedge\ \forall i : 1 \leq i < n. \\
&\qquad\qquad\qquad\qquad (source(\rho_i) = (source(t), \_) \\
&\qquad\qquad\qquad\qquad\qquad \wedge\ act(\rho_i)\ \textit{is hidden}))
\end{aligned}$$

and

$$\begin{aligned}
&\forall t \in T_Q.\ act(t) \in A^\star_Q \implies \\
&\qquad \exists t' \in T_{P\otimes Q}.\ (t' = ((source(t), \_), a, (target(t), \_)) \\
&\qquad\qquad \wedge\ (act(t), a) \in match(Q, P \otimes Q)) \\
&\quad \wedge\quad \forall v \in V_{P\otimes Q}.\ (v = (source(t), u)) \implies \\
&\qquad\qquad \exists path \rho = \{\rho_1, \ldots, \rho_n, \ldots\}.\ (\rho \in P \otimes Q \\
&\qquad\qquad\qquad \wedge\ source(\rho_1) = (source(t), u) \\
&\qquad\qquad\qquad \wedge\ act(\rho_n) = a \\
&\qquad\qquad\qquad \wedge\ \forall i : 1 \leq i < n. \\
&\qquad\qquad\qquad\qquad (source(\rho_i) = (source(t), \_) \\
&\qquad\qquad\qquad\qquad\qquad \wedge\ act(\rho_i)\ \textit{is hidden}))
\end{aligned}$$

where $\_$ is any state in the relevant component automaton.

4.2 Composition Examples

A simple example illustrates the role of master actions in composition. Consider the automata given in Fig. 2, assume a and b are shared actions and x is hidden (so not shared). In the composition, EX1 $\otimes$ EX2, master action $a!^\star$ waits for $a?^\circ$ at $(0, 0)$, i.e. they do not synchronise until $(0, 1)$. The slave $b!^\circ$ is never synchronised. If however $b!^\circ$ is replaced by $b!^\star$ in $EX1$, then EX1 $\otimes$ EX2 would be *invalid*.

As another example, consider the prediction server and storage server components described in Sect. 2, represented as pervasive interface automata in Fig. 3.

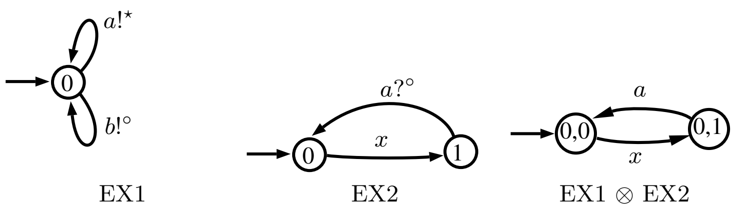


Fig. 2. Pervasive interface automata examples

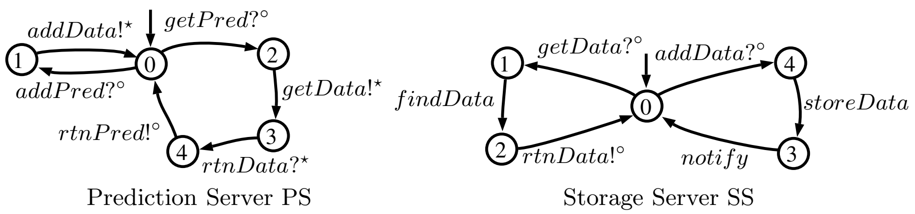


Fig. 3. Pervasive interface automata examples

The composition of these two automata will synchronise over the set of shared actions $\{getData, rtnData, addData\}$. The composition is shown in Fig. 4. Note the occurrence of a busy send in this example. From the product state (2,4), PS needs to be able to perform the master action *getData* but in state 4 SS is not yet ready to provide the matching slave action. Therefore, PS will then wait in state 2 until SS performs the *storeData* and *notify* actions before it returns to state 0 in which it is ready to perform the matching *getData* slave action. If SS was never enabled to receive the request, then the composition would be invalid.

5 Services

A service[1] is something that a component can do, such as respond to a data request, distribute information, store and retrieve data, etc; internal detail is not relevant. For example, a fixture server component P may offer the *getFixture* service, a service that always responds to the method call $getFix()$ by returning a list of fixtures via a return method $rtnFix()$. In component P, the service is offered in a straightforward way because the fixture list is held internally. Another component, P' say, may also offer the same service, even though it obtains the list of fixtures from a third component. For example, P' may respond to the $getFix()$ call by calling a third component to obtain the fixture list, which it then returns via the method $rtnFix()$. In both cases, P and P' offer the same *getFixture* service.

In terms of pervasive interface automata, a service can be described as a property defined in our own simple custom logic, defined below.

[1] Similar to a web service [1].

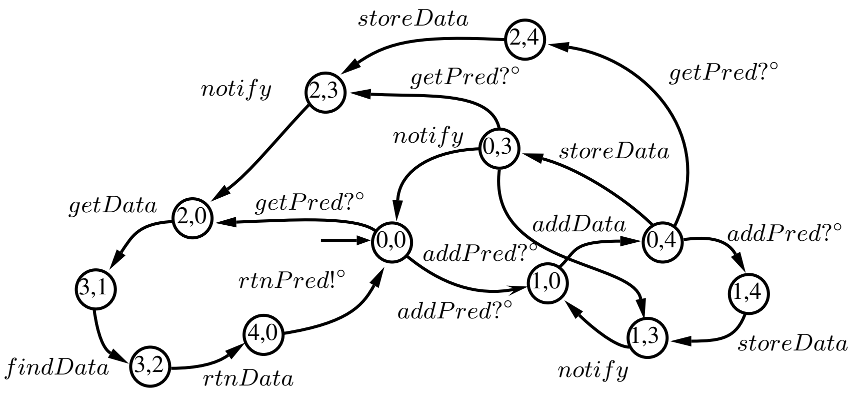


Fig. 4. The composition of PS and SS, PS $\otimes$ SS

5.1 Logic for Services

We now describe a linear temporal action logic for defining services (a simplification of that in [6]). The logic is defined over paths ρ of a pervasive interface automaton P. Here a and b are (annotated) actions.

Syntax:

$\phi = tt \mid \mathit{offer}\ a\ \phi \mid a \rightsquigarrow b\ \phi$	path formulae
$\Sigma = \forall\phi \mid \exists\phi \mid \Sigma \wedge \Sigma \mid \Sigma \vee \Sigma$	service formulae

Semantics:

$$
\begin{array}{lll}
\rho \models tt & \mathit{always} & \\
\rho \models \mathit{offer}\ a\ \phi & \text{iff } \exists n \geq 1.\ \rho = \{\ldots, t_n, \ldots\} & \\
 & \quad \text{and} & act(t_n) = a \\
 & \quad \text{and} & \forall_{i<n}\ act(t_i) \text{ are hidden actions} \\
 & \quad \text{and} & \{t_{n+1}, \ldots\} \models \phi \\
\rho \models a \rightsquigarrow b\ \phi & \text{iff } \exists n \geq 1, \exists m \geq n.\ \rho = \{\ldots, t_n, \ldots, t_m, \ldots\} & \\
 & \quad \text{and} & act(t_n) = a \\
 & \quad \text{and} & act(t_m) = b \\
 & \quad \text{and} & \forall_{i<n}\ act(t_i) \text{ are hidden actions} \\
 & \quad \text{and} & \forall_{n<i<m}\ act(t_i) \text{ are hidden actions} \\
 & \quad \text{and} & \{t_{m+1}, \ldots\} \models \phi \\
 & \text{or } \nexists n \geq 1.\ \rho\{\ldots, t_n, \ldots\} & \\
 & \quad \text{and} & act(t_n) = a \\
P \models \forall\phi & \text{iff } \forall\rho \in P.\ \ \rho \models \phi & \\
P \models \exists\phi & \text{iff } \exists\rho \in P.\ \ \rho \models \phi & \\
P \models \Sigma_1 \wedge \Sigma_2 & \text{iff } P \models \Sigma_1 \wedge P \models \Sigma_2 & \\
P \models \Sigma_1 \vee \Sigma_2 & \text{iff } P \models \Sigma_1 \vee P \models \Sigma_2 &
\end{array}
$$

Note that the actions preceding a are hidden in both *offer* $a\ \phi$ and $a \rightsquigarrow b\ \phi$. This expresses the requirement that the initiating action a of a service is available (e.g. cannot be blocked by waiting on another component), and if a service is initiated, then it is completed, e.g. a service is not abandoned.

5.2 Typical Services

A service often involves a response to a request (a liveness property). More precisely, after possible hidden actions, it offers the *request*, which is a callable method. After further possible hidden actions, it either *sends* a response, which is a method call, or it offers to *respond*, which is a callable method. The initial request may be accompanied by data (an input); the response may also be accompanied by data (an output).

Formally, assuming actions *request* (slave), *send* (master), and *respond* (slave), these two service are expressed in our logic by:

1. request/send: $\forall(\mathit{request}?^{\circ} \rightsquigarrow \mathit{send}!^{\star}\ \mathit{tt}) \wedge \exists(\mathit{offer}\ \mathit{request}?^{\circ}\ \mathit{tt})$
2. request/respond: $\forall(\mathit{request}?^{\circ} \rightsquigarrow \mathit{respond}!^{\circ}\ \mathit{tt}) \wedge \exists(\mathit{offer}\ \mathit{request}?^{\circ}\ \mathit{tt})$

The second conjunct: $\exists(\mathit{offer}\ \mathit{request}?^{\circ}\ \mathit{tt})$, serves to ensure the service is not trivially satisfied, i.e. the first action is offered by at least one path. As examples, the first type of service is offered by the prediction server PS (Fig. 3): $\mathit{addPred}?^{\circ}$ is always followed by $\mathit{addData}!^{\star}$, and the second is offered by the storage server SS: $\mathit{getData}?^{\circ}$ is always followed by $\mathit{rtnData}!^{\circ}$.

5.3 Components, Environments and Services

Our notion of a component automaton *offering* a service is not simply satisfaction of a service formula, we also take into account the impact and requirements placed upon the *environment*. We assume an environment is itself a composition of components. How to quantify the impact is subtle; it is not sufficient to check $P \models \Sigma$ (does a component P offer the service Σ), nor is it appropriate to check $P \otimes E \models \Sigma$ (does the composition of P with environment E offer Σ). In the former, Σ may not be satisfied because there are non-hidden actions that do not occur in Σ, but they will be hidden when the component is composed with the environment and in the latter, after composition, the actions occurring in Σ may have become hidden. Instead we consider an *abstraction* of environments that, if fulfilled, means that P offers the service Σ *to* E.

Specifically when checking if P offers the service Σ to E, we consider the availability of actions we require E to offer. We refer to the set of requirements for the availability of actions as A. The availability set A contains (action,state) pairs. A is split into two subsets A^+ and A^-. If $(a, s) \in A^+$ then the environment must offer action a in state s, if $(a, s) \in A^-$ then the environment must not offer action a in state s.

We define a set *cpath* of modified paths of a component automaton. There are two cases to consider. First, if an action (slave or master) can *always* be matched by the environment, then we assume the actions represent *hidden activity*. Second, if a slave action can *never* be matched with the environment, and it is offered from a component state that offers any other type of action (slave or hidden), then the slave action represents *spare capacity*. Note, we assume angelic non-determinism; this means that paths representing spare capacity will not be taken.

In the following, we write f°, g^* etc. to stand for any slave or master action respectively, that is we ignore input and output annotations. We call these abstract actions and when we say that abstract action f° matches f^*, we assume the underlying input and output annotations match as required. We refer to the set of actions in a service formula by $\alpha(\Sigma)$; for example, $\alpha(\forall a \rightsquigarrow\ b\ tt) = \{a, b\}$.

Definition 3. *The alphabet of a service formula, $\alpha(\Sigma)$, is the set of all actions occurring in the path sub-formulas of Σ.*

Definition 4. *Given component automaton P, service Σ and environment assumptions A, define the set $cpath(P, \Sigma, A^+, A^-)$, as all the paths of P constructed in the usual way* except, *when constructing the paths*

- *for transition t, if $act(t) = f^*$, $f^* \notin \alpha(\Sigma)$, and $(f^\circ, source(t)) \in A^+$, then replace $f^\star$ by f in t (hide master),*
- *for transition t, if $act(t) = f^\circ$, $f^\circ \notin \alpha(\Sigma)$, and $(f^\star, source(t)) \in A^+$, then replace f° by f in t (hide slave),*
- *for transition t, if $act(t) = f^\circ$,$f^\circ \notin \alpha(\Sigma)$, and $(f^\star, source(t)) \in A^-$, then any (sub)path beginning with t is excluded (remove spare capacity).*

Note that, under the conditions given above, some slave actions can be safely excluded from the set of paths. This is not the case for master actions as, by definition, a component requires to be able to perform them in order to function correctly.

As illustration of spare capacity and hidden activity, consider the two automata in Fig. 5 and the service $\Sigma = \forall(g1^\circ \rightsquigarrow g2^\circ\ tt)$. The alphabet of Σ is $\{g1^\circ, g2^\circ\}$. Both P_1 and P_2 include one hidden action, I.

The paths in $cpath(P_1, \Sigma, \{(l^\star, 2)\}, \{(f^\star, 1)\})$ start with either prefix $\rho_1 = \{h1^\circ, k^\circ, h2^\star\}$ or $\rho_2 = g1^\circ, I, l, g2^\circ\}$. As both $\rho_1 \models \Sigma$ and $\rho_2 \models \Sigma$, P_1 can offer

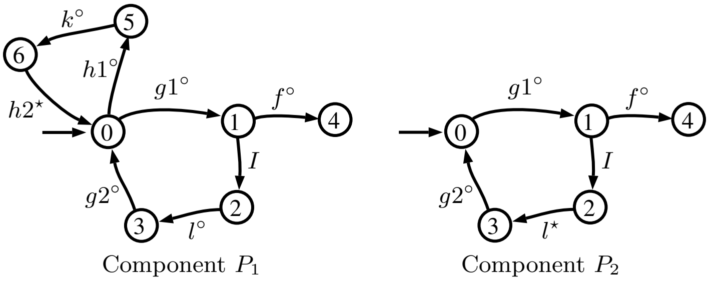


Fig. 5. Component automata, with abstract actions

Σ in an environment which offers $l^\star$ whenever P_1 is in state 2 and does not offer $f^\star$ when P_1 is in state 1. All the paths in $cpath(P_2, \Sigma, \{(l^\circ, 2)\}, \{(f^\star, 1)\})$ start with prefix $\rho_2 = \{g1^\circ, I, l, g2^\circ\}$. As $\rho_2 \models \Sigma$, P_2 can offer Σ in an environment which offers l° whenever P_2 is in state 2 and does not offer $f^\star$ when P_2 is in state 1.

Note that in both cases the path prefix $g1^\circ, f^\circ$ is excluded because the action f° is forbidden in state 1 (the environment will never offer a matching master action), whereas in component P_1, the path prefix $h1^\circ, k^\circ$ is included as spare capacity. In both cases the action l° has become hidden in *cpath*, because it matches an action offered by E.

In summary, the set *cpath* allows us to specify services that are offered by a component, in the context of an environment that meets assumptions A^+ and A^-. Together, A^+ and A^- are an abstraction of a class of environments. From here on we assume quantification over paths in $cpath(P, \Sigma, A^+, A^-)$ and we denote satisfaction with respect to A^+ and A^- by $\models_A$, defined thus.

Definition 5. *Given component automaton P, environment assumptions A, and path formula ϕ,*

$$\begin{array}{lll} P \models_A \forall\phi & \textit{iff} & \forall\rho \in cpath(P, \phi, A^+, A^-).\rho \models \phi \\ P \models_A \exists\phi & \textit{iff} & \exists\rho \in cpath(P, \phi, A^+, A^-).\rho \models \phi \\ P \models_A \Sigma_1 \wedge \Pi_2 & \textit{iff} & P \models_A \Sigma_1 \wedge P \models_A \Sigma_2 \\ P \models_A \Sigma_1 \vee \Pi_2 & \textit{iff} & P \models_A \Sigma_1 \vee P \models_A \Sigma_2 \end{array}$$

By abuse of notation we extend satisfaction to sets of services S, and write $P \models_A S$, when $\forall\Sigma \in S.P \models_A \Sigma$.

Note that P offers Σ to E can mean either P offers a service that E needs, or P offers a service that persists after P is composed with E. In the latter case we have $P \otimes E \models \Sigma$, but in the former case this is not true because the actions in Σ have become hidden.

5.4 Service Example

Consider again the *getFixture* service example. If we translate the offering of method *getFix*() as an action $getFix?^\circ$ and the *rtnFix*() method as action $rtnFix!^\circ$, we can express the *getFixture* service as the property:

$$getFixture = \forall(getFix?^\circ \rightsquigarrow rtnFix!^\circ\ tt) \wedge \exists(\mathit{offer}\ getFix?^\circ\ tt)$$

Figure 6 shows three example automata. The first, $FS1$, represents a component that can supply a fixture list without referring to any other component. $FS2$ represents a component that, upon request, requires a third party component (i.e. a storage server) before the requested fixture list can be returned. The third, $FS3$, offers the same basic functionality as $FS1$ (and thus offers the *getFix* service), however, it also offers the addition option of requesting a refined fixture list. More formally, we have:

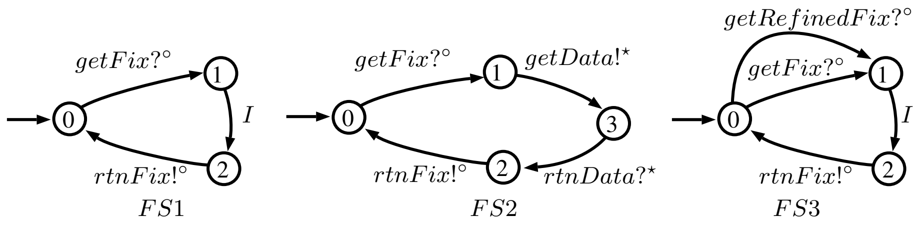


Fig. 6. Three example automata

- $FS1 \models_A getFixture$, where $A^+ = \emptyset$ and $A^- = \emptyset$.
- $FS2 \models_A getFixture$, where $A^+ = \{(getData?^\circ, 1), (rtnData!^\circ, 3)\}$ and $A^- = \emptyset$.
- $FS3 \models_A getFixture$, where $A^+ = \emptyset$ and $A^- = \emptyset$.

6 Replacement

Pervasive systems are adaptive and evolutionary by definition, meaning that components will be updated and replaced over time. Therefore, it is useful to know what effects such replacements will have on the system. Will the new component have the same functionality as the component it is replacing? To this end, we now define the replacement relation, which allows us to check if one component can be replaced by another in a given environment, with respect to a given set of services.

Definition 6. *For given component automata P, P', service Σ and environment requirements A, if $P \models_A \Sigma$, then P may be replaced by P' if $P' \models_{A'} \Sigma$ and $A' \subseteq A$.*

Replacement of a component with respect to a set of services is the natural extension of replacement with respect to a single service. This can be qualified: the *quality* of a replacement depends upon which services are preserved in a replacement, i.e.

Definition 7. *Given component automata P, P_1, P_2, sets of services SS, S_1, and S_2, and environment assumptions A, if $P \models_A SS$, $P_1 \models_A S_1$, and $P_2 \models_A S_2$, we say that component P_1 is a better replacement than component P_2 for P when $(S_2 \cap SS) \subset (S_1 \cap SS) \subseteq SS$.*

Note, we consider intersections of services, since the new components may offer additional services.

6.1 Replacement Example

Consider fixture server component $FS1$ from Fig. 6 that offers the *getFixture* service defined as $\forall(getFix?^\circ \leadsto rtnFix!^\circ) \wedge \exists(\mathit{offer}\ getFix?^\circ\ tt)$, with no environment assumptions. Which components could replace $FS1$?

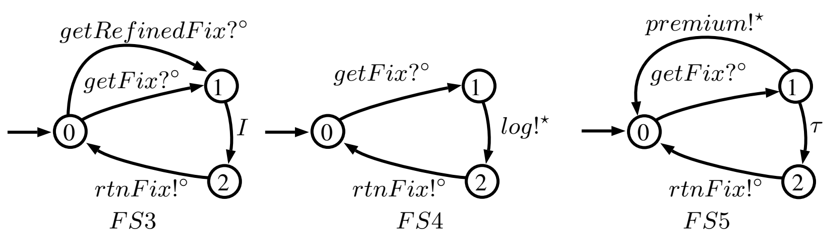


Fig. 7. Three potential replacement automata for FS1

Component $FS3$ offers the *getFixture* service with no environmental assumptions: $FS3 \models_A getFixture$ where $A^+ = \emptyset, A^- = \emptyset$. It also offers the additional option of requesting a refined fixture list. Therefore $FS3$ can replace $FS1$ in any environment; as $FS3$ offers additional functionality, this replacement would be referred to as an upgrade. Component $FS4$ also offers the *getFixture* service, however, it also logs each use of the service. Therefore, $FS4 \models_A getFixture$, where $A^+ = \{(log?^\circ, 1)\}$, $A^- = \emptyset$. That is, $FS4$ offers *getFixture* only if a logging service is available in the environment. So $FS4$ can only replace $FS1$ in environments offering a logging service. A fifth component, $FS5$, may be offered by some commercial organisation that requires a subscription to have access to some premium content. In which case, a $getFix?^\circ$ request may result in an error message notifying the environment that the premium content is inaccessible. As $premium!^\star$ is a master action, it cannot be ignored. Therefore, $FS5$ would not be a viable replacement for $FS1$, as $FS5 \not\models_A getFix$, for any A.

7 Comparison with Interface Automata and Session Types

Pervasive Interface Automata are based on Interface automata [3,4,5]. However, the addition classification of non-hidden actions as either master (*) or slave ($^\circ$), results in a richer action set. This combined with the more relaxed definition of composability of pervasive interface automata make them more appropriate for the context of pervasive systems.

A state $s = (f, g)$ in the composition of two interface automata F and G is said to be an *error state* if there is a shared action a that is an output action in F and enabled in f, but the corresponding action is not enabled in g. This disallows a component to wait for another to be ready before it performs an output action. Such conditions are prevalent and desirable in pervasive systems. Our notion of master and slave actions allow us to both define and embrace this kind of behaviour.

A crucial aspect of pervasive interface automata is that they allow us to formally define the notion of *replacement* of components, with respect to services. This concept relies heavily on our categorisation of actions as either master or slave, and so is not possible with interface automata.

The stated objectives in [11] are similar to our own, however the authors do not differentiate between master and slave actions. They also have a less rich definition of services.

Pervasive interface automata (and indeed interface automata) also bear a superficial resemblance to *session types* [8,9]. A session represents possible sequences of communication events between processes. Communication events include synchronous message passing and the passing of channel names. Again there is no notion of master/slave actions, and properties are restricted to the matching of communication events.

8 Conclusion and Future Work

We have introduced pervasive interface automata as a formalism for modelling interfaces offered by components. Our motivation is managing predictability in component-based systems, especially in pervasive systems where components are regularly composed and replaced. Distinctive features of our automata include the separation of actions according to input or output, and method call or callable method. Composition of automata involves synchronisation on input/output and calling/called actions.

We do not just model interfaces, but also reason about services, which we define using a linear temporal action logic. We define the notion of a component offering a service *to* an environment (a composition of components) by considering the assumptions we need to make about the environment. If an environment meets those assumptions, then we can be assured that either the component meets the (service) needs of the environment or the service will persist after composition. In either case, we can proceed with composition. A key relation is replacement, which may add new functionality, but ensures that services are still offered, given environment assumptions. Key concepts are illustrated with a mobile phone based application for sports predictions. Our long term goal is to derive pervasive interface automata automatically from code; this is future work.

References

1. Alonso, G., Casati, F., Kuno, H., Machiraju, V.: Web Services: Concepts, Architectures and Applications. Data-centric systems and applications. Springer, Heidelberg (2004)
2. Bell, M., Hall, M., Chalmers, M., Gray, P., Brown, B.: Domino: Exploring mobile collaborative software adaptation. In: Fishkin, K.P., Schiele, B., Nixon, P., Quigley, A. (eds.) PERVASIVE 2006. LNCS, vol. 3968, pp. 153–168. Springer, Heidelberg (2006)
3. de Alfaro, L., Henzinger, T.: Interface automata. SIGSOFT Software Engineering Notes 26(5), 109–120 (2001)
4. de Alfaro, L., Henzinger, T.: Interface theories for component-based design. In: Henzinger, T., Kirsch, C. (eds.) EMSOFT 2001. LNCS, vol. 2211, pp. 148–165. Springer, Heidelberg (2001)

5. de Alfaro, L., Henzinger, T.: Interface-based design. Engineering Theories of Software-intensive Systems 195, 83–104 (2005)
6. de Nicola, R., Vaandrager, F.: Action versus state based logics for transition systems. In: Guessarian, I. (ed.) LITP 1990. LNCS, vol. 469, pp. 407–419. Springer, Heidelberg (1990)
7. Google: App inventor for android (July 2010), `http://appinventor.googlelabs.com`
8. Honda, K.: Types for dyadic interaction. In: Best, E. (ed.) CONCUR 1993. LNCS, vol. 715, pp. 509–523. Springer, Heidelberg (1993)
9. Honda, K., Vasconcelos, V., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: Hankin, C. (ed.) ESOP 1998. LNCS, vol. 1381, pp. 122–138. Springer, Heidelberg (1998)
10. Milner, R.: A Calculus of Communication Systems. LNCS, vol. 92. Springer, Heidelberg (1980)
11. Černá, I., Vařeková, P., Zimmerova, B.: Component substitutability via equivalencies of component-interaction automata. Electronic Notes in Theoretical Computer Science (ENTCS) 182, 39–55 (2007)

A Separation Logic for OO Programs[⋆]

Liu Yijing and Qiu Zongyan

LMAM and Department of Informatics, School of Mathematical Sciences, Peking University
{liuyijing,qzy}@math.pku.edu.cn

Abstract. We present a general storage model that reflects features of object oriented (OO) languages with pure reference semantics. Based on this model, we develop an OO Separation Logic (OOSL) to specify and verify OO programs. Many inference rules in the Separation Logic still hold in OOSL. Additionally, OOSL has certain properties important to OO reasoning. We introduce Hoare-Triple for a small OO language, and use the Schorr-Waite Marking Algorithm as a verification example.

Keywords: Object Orientation, Separation Logic, Verification.

1 Introduction

Object-orientation (OO) paradigm is and will remain important for software development and programming languages design, because it supports many very useful abstractions. However, many new challenges in program specification and verification present in the OO field. There are two key issues mutually depending on each other: (1) building proper formal models for OO languages, and (2) developing useful methods to specify and verify OO programs. Researchers have proposed many formal frameworks to describe core concepts of OO programs.

As the basis for formal studies, various state models are proposed to represent complicated structures of state space of OO programs. Major models can be roughly classified as *Object Graph Model*, *Access Trace Model*, and *Stack Heap Model*.

The Object Graph Models treat objects state as some form of graphs. Examples in this direction include the topological model [12], or object diagram [17]. In the graph, vertexes denote objects, and edges denote variables and object attributes (i.e., instance variables). Models of this kind are intuitive and always independent of languages. [6] presents an operational semantics based on a graph model. However, a suitable reasoning framework for graph models still does not exist. The Access Trace Model (originally for pointer-programs) was introduced by [4], where each object was identified by a set of traces to the object. Access Trace models have advantages in alias analysis [2], but seem too abstract for general purpose. [3] attempted to define a general inference framework for a trace model. Stack Heap Models are extensions of normal store model, with an additional heap (a map from address to values) to represent objects. Stack Heap Models seem low-level, however, they are relatively easy to used for definitions of program semantics. Some works have been done upon such models, e.g. [13]. However, a full accounting of all important OO features is still missing.

[⋆] Supported by NNSFC Grant No. 60718002 and No. 60773161.

On the other hand, many works have been done on the specification and verification of OO programs. Although JML [7] and Spec# [1] catch increasing attentions, many critical issues of OO programs, especially that related to the mutable object structures, have not been considered deeply there. Many semantic issues must be investigated and understood for a big-leap in this field. [8] gives a comprehensive overview for the achievements and challenges in this area.

Separation Logic [15] is a powerful tool to handle shared mutable data structures. Many verification techniques based on it have been developed, mainly for C-like programs, and some targeting OO programs. However, it is not straightforward to use the Separation Logic to specify and verify OO programs, because the underlying storage model of the logic is not ready for many OO concepts. Especially, there is no correspondents of object attributes in the model. Some researchers tried to revise the Separation Logic targeting OO fields [9,14]. The work presented here is also in this direction. We will discuss and compare these works in Section 6.

In this article, we proposed a model for the object pool (heap), with a novel definition for the separation of object pools. The model provides a clear concept for objects. Empty objects can be naturally represented and reasoned. We develop a revised Separation Logic, named OO Separation Logic (OOSL), for expressing OO program states. User-defined predicates and logic environment are clearly defined, and the semantics of the logic is defined as a least fix-point which is guaranteed existing. The logic adopts classical semantics, thus is more expressive than the logic with intuitionistic semantics (referring to [5]) as used in [14]. Properties of OOSL are explored, especially a new concept named *separated assertions*, which is useful in reasoning OO programs. Due to the classical nature, properties of OO programs can be precisely specified and verified. We introduce a simple OO language, and develop Hoare-Triple like inference rules based on OOSL. We use Schorr-Waite Marking Algorithm as an example to show how to specify and verify OO programs with OOSL.

The rest of the paper is organized as follows: We introduce an OO storage model in Section 2. OOSL is developed in Section 3. We introduce a simple OO language and its inference rules in Section 4. In Section 5, we study the Schorr-Waite Marking Algorithm. Finally, we discussed some related works and future research directions.

2 An OO Storage Model

Now we introduce a storage model for OO programs with pure reference nature.

The model is defined based on three basic sets Name, Type and Ref, these sets model basic concepts, such as variables, fields, types and references, in OO program.

- Name: an infinite set of names, used for naming various entities, e.g., constants, variables, attributes, etc. Three special names, $\mathbf{true}, \mathbf{false}, \mathbf{null} \in \mathrm{Name}$, denote boolean and null constants.
- Type: an infinite set of types, including predefined types and user-defined types (or called classes). Subtype relation is represented by symbol $<:$, where $T_1 <: T_2$ states that T_1 is a subtype of T_2. We assume there are three predefined types $\mathbf{Object}, \mathbf{Null}$ and $\mathbf{Bool}$. $\mathbf{Object}$ is the super type of all classes. $\mathbf{Null}$ is the subtype of all classes. And $\mathbf{Bool}$ is the type of boolean objects. Given a type T,

we can obtain its attributes by function $\mathsf{attrs} : \mathrm{Type} \rightarrow \mathrm{Name} \rightarrow \mathrm{Type}$; and we define $\mathsf{attrs}(\mathbf{Object}) = \mathsf{attrs}(\mathbf{Null}) = \mathsf{attrs}(\mathbf{Bool}) = \emptyset$. Other predefined types, such as Integer, can be added easily, but we consider only boolean type here.

– Ref: an infinite set of references which are the identities of objects. Corresponding to the Name constants, Ref contains three basic references rtrue, rfalse and rnull, where rtrue, rfalse refer two $\mathbf{Bool}$ objects, and rnull never refers to any object. We assume two primitive functions on Ref:[1]
  - $\mathsf{eqref} : \mathrm{Ref} \rightarrow \mathrm{Ref} \rightarrow \mathsf{bool}$, justifies whether two references are same, i.e. given references $r_1, r_2 \in \mathrm{Ref}$, $\mathsf{eqref}(r_1, r_2) = \mathsf{true}$ iff r_1 is same to r_2.
  - $\mathsf{type} : \mathrm{Ref} \rightarrow \mathrm{Type}$, decides the runtime type of object referred by reference. We define $\mathsf{type}(\mathsf{rtrue}) = \mathsf{type}(\mathsf{rfalse}) = \mathbf{Bool}, \mathsf{type}(\mathsf{rnull}) = \mathbf{Null}$.

In fact, Name, Type and functions (relations) defined on them, such as dtype, $<:$ and attrs, make up the static information of an OO program.

Based on above concepts, we define an OO storage model. It is similar to the classical Stack-Heap model with two components:

$$\mathrm{Store} \mathrel{\widehat{=}} \mathrm{Name} \rightharpoonup_{\mathsf{fin}} \mathrm{Ref} \qquad \mathrm{Opool} \mathrel{\widehat{=}} \mathrm{Ref} \rightharpoonup_{\mathsf{fin}} \mathrm{Name} \rightharpoonup_{\mathsf{fin}} \mathrm{Ref}$$
$$\mathrm{State} \mathrel{\widehat{=}} \mathrm{Store} \times \mathrm{Opool}$$

where notation "$\rightharpoonup_{\mathsf{fin}}$" denotes finite partial functions.

We will use σ and O, possibly with subscript, to denote elements of Store and Opool respectively. A store $\sigma \in \mathrm{Store}$ maps variables and constants to references, and an object pool $O \in \mathrm{Opool}$ maps references to field-reference pairs. A runtime state s is a pair, $s = (\sigma, O) \in \mathrm{State}$, consisting of a store and an object pool. For every $\sigma \in \mathrm{Store}$, we assume that $\sigma\mathbf{true} = \mathsf{rtrue}$, $\sigma\mathbf{false} = \mathsf{rfalse}$ and $\sigma\mathbf{null} = \mathsf{rnull}$.

We will use $r, r_1, \ldots$ to denote references, and $a, a_1, \ldots$ to denote attributes of objects. An element of O is a pair (r, f), where r is a reference to some object o, f is a function from attributes of o to their corresponding values (also references). When we mention the domain of O, we sometimes want to mean a subset of Ref associated with a set of objects as discussed above, or sometimes a subset of $\mathrm{Ref} \times \mathrm{Name}$ associated with a set of values (references). We use $\mathsf{dom}\, O$ for the first case. For the second case, we define notation $\mathsf{dom}_2\, O \mathrel{\widehat{=}} \{(r, a) \mid r \in \mathsf{dom}\, O, a \in \mathsf{dom}\, O(r)\}$, that is, $\mathsf{dom}_2\, O$ gets all the reference and attribute pairs of non-empty objects in O.

When considering the program states, we need to ask for some regularity, that is, the well-typedness. Now we define the concept for states that are consistent with the static information of the program, i.e., the well-typed states. We assume a function $\mathsf{dtype} : \mathrm{Store} \rightarrow \mathrm{Name} \rightarrow \mathrm{Type}$, where $\mathsf{dtype}(\sigma)(v)$ gives the declaration type of constant or variable v in store σ.

Definition 1 (Well-typed Store). *A store σ is well-typed iff*

$$\forall v \in \mathsf{dom}\, \sigma \cdot \mathsf{type}(\sigma(v)) <: \mathsf{dtype}(\sigma)(v).$$

Clearly, this condition requires that all variables hold valid values.

[1] For example, we can define every reference as a pair (t, id) where $t \in \mathrm{Type}$ and $id \in \mathbf{N}$, define eqref as pair equivalence, and $\mathsf{type}(r) = r.first$.

Definition 2 (Well-typed Opool). *An Opool O is well-typed iff*

- $\forall (r, a) \in \mathsf{dom}_2\, O \cdot a \in \mathsf{Att}(r) \wedge \mathsf{type}(O(r)(a)) <: \mathsf{attrs}(r)(a)$, and
- $\forall r \in \mathsf{dom}\, O \cdot \mathit{Att}(r) = \emptyset \vee (\mathsf{Att}(r) \cap \mathsf{dom}\, O(r) \neq \emptyset)$.

where $\mathsf{Att}(r) \mathrel{\widehat{=}} \mathsf{dom}\, \mathsf{attrs}(\mathsf{type}(r))$.

Note that $\mathsf{attrs}(\mathsf{type}(r))$ is a function from an attribute set to Type. The first condition requires that all attributes are valid according to types of all objects in O, and all attributes hold values of correct types. The second condition requires that if a non-empty object (according to its type) is in O, then O must contains at least one attribute of the object. Thus we can identify empty objects in any Opool.

As an example, suppose we have statically $\mathsf{dom}\, \mathsf{attrs}(C) = \{a_1, a_2, a_3\}$, and have a program state where $\mathsf{type}(r_1) = \mathbf{Object}, \mathsf{type}(r_2) = C$. In this case, we easily know that $O_1 = \{r_1 \mapsto \emptyset, r_2 \mapsto \{a_1 \mapsto \mathsf{rnull}, a_2 \mapsto \mathsf{rnull}\}\}$ is a well-typed Opool, but $O_2 = \{r_1 \mapsto \emptyset, r_2 \mapsto \emptyset\}$ is not, because $\mathsf{type}(r_2) = C$ has attributes. Further, we can calculate that $\mathsf{dom}\, O_1 = \{r_1, r_2\}$, and $\mathsf{dom}_2\, O_1 = \{(r_2, a_1), (r_2, a_2)\}$.

Definition 3 (Well-typed State). *A state $s = (\sigma, O)$ is well-typed iff both σ and O are well-typed.*

We will only consider well-typed states from now on. This requirement makes sense because a well-typed program always runs under well-typed states, and the well-typedness can be checked statically based on the type system of the language.

For convenience, we will use notation $(r, \{\overline{(a, \text{-})}\})$ to denote an (or a part of an) object, and use $(r, a, \text{-})$ to denote a cell (state of an attribute of an object) in the Opool. Here "-" represents some value which we do not care about.

We define a special overriding operator $\oplus$ on Opool:

$$(O_1 \oplus O_2)(r) \mathrel{\widehat{=}} \begin{cases} O_1(r) \oplus O_2(r) & \text{if } r \in \mathsf{dom}\, O_2 \\ O_1(r) & \text{otherwise} \end{cases}$$

where the right $\oplus$ is the standard function overriding operator. Thus, for Opool O_1, $O_1 \oplus \{(r, a, r')\}$ gives a new Opool, where only one attribute value (the value for a) of the object pointed by r is modified (denoted by r').

We borrow some concepts and notations from the Separation Logic. $O_1 \perp O_2$ indicates that two Opools O_1 and O_2 are separated from each other. The formal definition for $\perp$ is new for separating object pools,

$$\begin{aligned} O_1 \perp O_2 \mathrel{\widehat{=}} {} & \forall r \in \mathsf{dom}\, O_1 \cap \mathsf{dom}\, O_2 \cdot \\ & \quad O_1(r) \neq \emptyset \wedge O_2(r) \neq \emptyset \wedge \mathsf{dom}\,(O_1(r)) \cap \mathsf{dom}\,(O_2(r)) = \emptyset. \end{aligned}$$

That is, if a reference, referring to some object o, is in both $\mathsf{dom}\, O_1$ and $\mathsf{dom}\, O_2$, then both O_1 and O_2 must contain non-empty subsets of o's attributes, respectively (the well-typedness also guarantees this); and these two subsets must be disjoint. This means that we can separate attributes of an object in the Opool (providing that the object is not empty). Additionally, an empty object cannot be in two separated Opools at the same time, because it cannot be partitioned. We will use $O_1 * O_2$ to indicate the union of O_1 and O_2, $O_1 \oplus O_2$, when $O_1 \perp O_2$.

As an example, suppose,

$$O_1 = \{(r_1, \emptyset), (r_2, \{(a_1, \mathsf{rnull})\})\}, \;\; O_2 \;=\; \{(r_2, \{(a_2, \mathsf{rnull})\})\},$$
$$O_3 = \{(r_1, \emptyset), (r_2, \{(a_2, \mathsf{rnull})\})\}.$$

We have $O_1 \perp O_2$, although each of them contains a part of object pointed by r_2. But $O_1 \not\perp O_3$ because $r_1 \in \mathsf{dom}\, O_1 \cap \mathsf{dom}\, O_3$, while $O_1(r_1) = \{\}$. Additionally, $O_2 \not\perp O_3$, because $r_2 \in \mathsf{dom}\, O_2 \cap \mathsf{dom}\, O_3$, while $\mathsf{dom}\,(O_2(r_2)) \cap \mathsf{dom}\,(O_3(r_2)) = \{a_2\}$.

Clearly, above definition of separation takes the basic cell (r, a, r') as a unit, but it also offers a careful treatment for empty objects. It is a revision of the separation concept in Separation Logic, while also takes into account the characteristics of OO programs. This definition plays an important role in our work.

3 An OO Separation Logic

To facilitate OO features, almost all OO languages adopt pure reference models, where values of variables and object attributes are references to objects[2]. A special case is that their values can be null to mean referring to no object. This model induces a great possibility of sharing: besides different variables can share references, different attributes can also share references, and can have sharing with variables. For modeling these features, we define an OO Separation Logic(OOSL) for OO specialities.

3.1 Assertions Language

We use Ψ for the set of all assertions of OOSL, and $\psi, \psi_1, \psi_2 ...$ as typical assertions. The assertion language of OOSL is similar to what in Separation Logic, with some revisions and extensions, to fit the special needs of OO programs.

Basic assertions are of two kinds in OOSL, namely *primitive assertions* and *user-defined assertions*. All assertions are built on them.

Primitive assertions have the forms defined by the following rules:

$$\alpha ::= \mathbf{true} \mid \mathbf{false} \mid v = r \mid r_1 = r_2$$
$$\beta ::= \mathbf{emp} \mid r_1.a \mapsto r_2 \mid \mathsf{obj}(r, T)$$

where v is a variable or constant name, r denotes references. In fact, here r servers as both "references" and "reference variables" (logic variables) at the same time.

As shown, primitive assertions fall into two categories, where

- α denotes a kind of assertions that are independent of Opools. References are atomic values in our logic. For any two references r_1, r_2, $r_1 = r_2$ holds iff r_1 and r_2 are identical, i.e., $\mathsf{eqref}(r_1, r_2)$. We treat $r = v$ the same as $v = r$.
- β denotes assertions involving Opools. Empty and singleton assertions take the similar forms as in Separation Logic. As we said before, a cell in Opool is an attribute-value binding of an object (denoted by a reference), thus the singleton

[2] One exception might be variables and attributes of primitive types, while many languages use value model for them for efficiency.

assertion takes the form $r_1.a \mapsto r_2$. To make OOSL clear and simple, we do not define $v.a \mapsto \ldots$ as a primitive assertion, because it is not really primitive. Certainly, we can define $v.a \mapsto r$ as $\exists r' \cdot v = r' \wedge r.a \mapsto r'$.

– We add an assertion form $\mathsf{obj}(r, T)$ to indicate that r refers to a complete object of type T, and the Opool only contains this object. In Separation Logic, people use $l \mapsto -$ or $l \hookrightarrow -$ to denote that location l is allocated in current heap. Because the existence of empty object, we cannot use $r.a \mapsto -$ or $r.a \hookrightarrow -$ to express that object which r refers to is allocated in current Opool. To solve this problem, we introduce assertion form $\mathsf{obj}(r, T)$ in OOSL. We will use $\mathsf{obj}(r, -)$ when we do not care about r's type.

We allow users to define new predicates in OOSL. In fact, people always need to define some recursive predicates to support specification and verification of OO programs involving recursive data structures, e.g., list, tree, etc.

These definitions are recorded in a *Logic Environment* Λ with the form defined by:

$$\Lambda ::= \varepsilon \mid \Lambda, p(\overline{r}) \doteq \psi$$

where ε denotes the empty environment, p is a symbol (predicate name) selected from a given set $\mathcal{S}$, $\overline{r}$ are (a list of) formal parameters, and ψ is the body, which is an assertion correlated with $\overline{r}$. Recursive definitions are allowed.

As a well-formed logic environment, we ask for that Λ must be self-contained, that is: The body ψ of a definition in Λ cannot use symbols not defined in Λ. Further, we require that Λ must be *finite* and *syntactically monotone*[3], then a fix-point semantics for Λ exists.

For every symbol p defined in Λ, we use $\mathsf{argc}_\Lambda(p)$ to denote its arguments number, where subscript Λ may be omitted when there is no ambiguity.

Complex assertions are built upon basic assertions with classical FOL combinators and separation combinators from Separation Logic:

$$\psi ::= \alpha \mid \beta \mid p(\overline{r}) \mid \neg\psi \mid \psi \vee \psi \mid \psi * \psi \mid \psi -\!\!* \psi \mid \exists r \cdot \psi$$

where $p(\overline{r})$ is a user-defined assertion with real arguments $\overline{r}$.

Please notice that only references, but not variables, can be quantified. The intension is clear: variables are defined in the program text, thus are free variables in assertions.

We will use $\psi[v/x]$ (or $\psi[r/x]$) to denote the assertion built from ψ by substituting variable x with variable or constant v (reference r) in it. And $\psi[r_1/r_2]$ denotes the assertion build from ψ by substituting r_2 with r_1.

At last, we define some abbreviations, that are classical:

$$\begin{array}{ll} \psi_1 \wedge \psi_2 \equiv \neg(\neg\psi_1 \vee \neg\psi_2) & \psi_1 \Rightarrow \psi_2 \equiv \neg\psi_1 \vee \psi_2 \\ \forall r \cdot \psi \quad\; \equiv \neg\exists r \cdot \neg\psi & \\ r.a \mapsto - \;\equiv \exists r' \cdot r.a \mapsto r' & r.a \hookrightarrow r' \equiv r.a \mapsto r' * \mathbf{true} \end{array}$$

The last two abbreviations are widely used in Separation Logic related papers.

[3] For every definition $p(\overline{r}) \doteq \psi$, every symbol occurs in ψ must lie under an even number of negations.

$$
\begin{array}{lll}
\mathcal{M}_{\mathcal{I}}(\mathbf{false}) & = \emptyset & \text{(I-FALSE)} \\
\mathcal{M}_{\mathcal{I}}(\mathbf{true}) & = \mathrm{State} & \text{(I-TRUE)} \\
\mathcal{M}_{\mathcal{I}}(v = r) & = \{(\sigma, O) \mid \sigma(v) = r\} & \text{(I-LOOKUP)} \\
\mathcal{M}_{\mathcal{I}}(r_1 = r_2) & = \mathrm{State} \quad \text{iff } \mathsf{eqref}(r_1, r_2) & \text{(I-REF-EQ)} \\
\mathcal{M}_{\mathcal{I}}(r_1 = r_2) & = \emptyset \quad \text{iff } \neg\mathsf{eqref}(r_1, r_2) & \text{(I-REF-NEQ)} \\
\mathcal{M}_{\mathcal{I}}(\mathbf{emp}) & = \{(\sigma, \emptyset)\} & \text{(I-EMPTY)} \\
\mathcal{M}_{\mathcal{I}}(r_1.a \mapsto r_2) & = \{(\sigma, \{(r_1, a, r_2)\})\} & \text{(I-SINGLE)} \\
\mathcal{M}_{\mathcal{I}}(\mathsf{obj}(r, T)) & = \{(\sigma, O) \mid \mathsf{type}(r) = T \wedge \mathsf{dom}\, O = \{r\} \wedge & \text{(I-OBJ)} \\
 & \qquad \mathsf{dom}\,(O(r)) = \mathsf{dom}\,(\mathsf{attrs}(T))\} & \\
\mathcal{M}_{\mathcal{I}}(p(\overline{r})) & = \mathcal{I}(p)(\overline{r}) & \text{(I-APP)} \\
\mathcal{M}_{\mathcal{I}}(\neg\psi) & = \mathrm{State} \setminus \mathcal{M}_{\mathcal{I}}(\psi) & \text{(I-NEG)} \\
\mathcal{M}_{\mathcal{I}}(\psi_1 \vee \psi_2) & = \mathcal{M}_{\mathcal{I}}(\psi_1) \cup \mathcal{M}_{\mathcal{I}}(\psi_2) & \text{(I-OR)} \\
\mathcal{M}_{\mathcal{I}}(\psi_1 * \psi_2) & = \{(\sigma, O) \mid \exists O_1, O_2 \cdot O_1 * O_2 = O \wedge (\sigma, O_1) \in \mathcal{M}_{\mathcal{I}}(\psi_1) & \text{(I-S-CONJ)} \\
 & \qquad \wedge (\sigma, O_2) \in \mathcal{M}_{\mathcal{I}}(\psi_2)\} & \\
\mathcal{M}_{\mathcal{I}}(\psi_1 -\!\!* \psi_2) & = \{(\sigma, O) \mid \forall O_1 \cdot O_1 \perp O \wedge (\sigma, O_1) \in \mathcal{M}_{\mathcal{I}}(\psi_1) & \text{(I-S-IMPLY)} \\
 & \qquad \text{implies } (\sigma, O_1 * O) \in \mathcal{M}_{\mathcal{I}}(\psi_2) & \\
\mathcal{M}_{\mathcal{I}}(\exists r \cdot \psi) & = \{(\sigma, O) \mid \exists r \in \mathrm{Ref} \cdot (\sigma, O) \in \mathcal{M}_{\mathcal{I}}(\psi)\} & \text{(I-EX)}
\end{array}
$$

Fig. 1. Semantic function with interpretation $\mathcal{I}$

3.2 Semantics

Now, we provide a *Least Fix-point Semantics* for OOSL. We will define a semantic function which maps every assertion $\psi \in \Psi$ to a subset of State. To achieve this goal, we first define a formal semantics for Λ.

We introduce a family of *Predicate Functions*. For any $n \geq 0$, we define $\mathcal{P}_n \mathrel{\widehat{=}} \mathrm{Ref}^n \to \mathbb{P}(State)$, the set of functions from n references to subsets of State. Here n is the arity of the functions in $\mathcal{P}_n$. We define $\mathcal{P} \mathrel{\widehat{=}} \bigcup_n \mathcal{P}_n$, which is the set of all possible predicate functions. We introduce a function $\mathsf{arity} : \mathcal{P} \to \mathbf{N}$ to extract the arity of given predicate function: For any $p \in \mathcal{P}$, $\mathsf{arity}(p) = n$ iff $p \in \mathcal{P}_n$.

We will use p, q, possibly with subscripts, for the typical elements of $\mathcal{P}$. Given $p(\overline{r}), q(\overline{r'}) \in \mathcal{P}_n$, we define $p \leq q$ iff $\forall r_1, ..., r_n \cdot p(r_1, ..., r_n) \subseteq q(r_1, ..., r_n)$. Clearly, $(\mathbb{P}(State), \subseteq)$ forms a complete lattice, with $\emptyset$ and State as its bottom and top elements. So for any n, $(\mathcal{P}_n, \leq)$ is a complete lattice, with $\bot_{\mathcal{P}_n} = \{(r_1, ...r_n) \mapsto \emptyset\}, \top_{\mathcal{P}_n} = \{(r_1, ...r_n) \mapsto \mathrm{State}\}$ as its bottom and top elements.

With Predicate Functions, we define interpretations of Λ as follows.

Definition 4 (Interpretation of Logic Environment). *Given a logic environment Λ, we say a function $\mathcal{I} : \mathcal{S} \to \mathcal{P}$ is an interpretation of Λ iff for every symbol* p *defined in Λ, $p \in \mathsf{dom}\,\mathcal{I}$ and $\mathsf{arity}(\mathcal{I}(p)) = \mathsf{argc}_\Lambda(p)$.*

We use $\mathcal{I}_\Lambda$ to denote all interpretations of Λ. For any $\mathcal{I}_1, \mathcal{I}_2 \in \mathcal{I}_\Lambda$, we define:

$$\mathcal{I}_1 \leq \mathcal{I}_2 \text{ iff } \forall p \in \mathsf{dom}\,\Lambda \cdot \mathcal{I}_1(p) \leq \mathcal{I}_2(p).$$

Obviously, $(\mathcal{I}_\Lambda, \leq)$ is a complete lattice. $\bot_\Lambda = \{(p, \bot_{\mathcal{P}_{\mathsf{argc}_\Lambda(p)}}) | p \in \mathsf{dom}\,\Lambda\}$ is the bottom element, and $\top_\Lambda = \{(p, \top_{\mathcal{P}_{\mathsf{argc}_\Lambda(p)}}) | p \in \mathsf{dom}\,\Lambda\}$ is the top element.

We define a semantic function $\mathcal{M} : \mathcal{I} \to \Psi \to \mathbb{P}(\mathrm{State})$ for OOSL, the definition is presented in **Fig.1**. Note that $\mathcal{M}_\mathcal{I}$ means $\mathcal{M}(\mathcal{I})$ in the definition.

Clearly, a logic environment Λ can have many different interpretations, but not every interpretation makes sense. This leads the following definition.

Definition 5 (Model of Logic Environment). *Suppose $\mathcal{I}$ is an interpretation of Λ, we say $\mathcal{I}$ is a model of Λ iff for every definition $p(\overline{r}) \doteq \psi$ in Λ, we have:*

$$\forall \overline{r'} \cdot \mathcal{M}_\mathcal{I}(p(\overline{r'})) = \mathcal{M}_\mathcal{I}(\psi[\overline{r'}/\overline{r}]).$$

In fact, a model of Λ is a fix-point of function $\mathcal{N}_\Lambda : (\mathcal{S} \to \mathcal{P}) \to (\mathcal{S} \to \mathcal{P})$, which is defined as follows:

$$\mathcal{N}_\Lambda(\mathcal{I})(p) = \{(\overline{r'}, \mathcal{M}_\mathcal{I}(\psi[\overline{r'}/\overline{r}])\}, \quad \text{for any definition } p(\overline{r}) \doteq \psi \text{ in } \Lambda$$

The fix-point of $\mathcal{N}_\Lambda$ exists, because the self-containedness of Λ, and the syntactically monotonic requirement for each definition of symbols in Λ.

A given Λ may have many models. We choose the least one as its standard model, which is the *least fix-point* of $\mathcal{N}$. By Tarski's fix-point theorem, this standard model can be expressed as:

$$\mathcal{J}_\Lambda = \bigcup_{n=0}^{\infty} \mathcal{N}_\Lambda^n(\bot_\Lambda),$$

We give a simple example as an illustration. Suppose Λ contains only one definition

$$list(r) \doteq (r = \mathbf{null} \wedge \mathbf{emp}) \vee \exists r' \cdot (r.a \mapsto r') * list(r')$$

which describes lists linked on a. In order to get the standard model of Λ, we have:

$$\begin{array}{l}
\mathcal{N}_\Lambda^0 = \bot_\Lambda \\
\mathcal{N}_\Lambda^1 = \{(list, \{(\mathbf{null}, \mathbf{emp})\})\} \\
\mathcal{N}_\Lambda^2 = \{(list, \{(\mathbf{null}, \mathbf{emp}), (r, r.a \mapsto \mathbf{null})\})\} \\
\mathcal{N}_\Lambda^3 = \{(list, \{(\mathbf{null}, \mathbf{emp}), (r, r.a \mapsto \mathbf{null})\}), (r, r.a \mapsto r' * r'.a \mapsto \mathbf{null})\})\} \\
\ldots
\end{array}$$

Then we get a model that describes all possible lists of this type.

With the standard model $\mathcal{J}_\Lambda$, we can define the formal semantics for our assertion language. We use $\sigma, O \models_\Lambda \psi$ to mean that ψ holds on state (σ, O) with respect to logic environment Λ. We have the following definition:

Definition 6 (Semantics of Assertions)

$$\sigma, O \models_\Lambda \psi \qquad \mathit{iff} \qquad (\sigma, O) \in \mathcal{M}_{\mathcal{J}_\Lambda}(\psi).$$

We often use $\sigma, O \models \psi$ as a shorthand when Λ is not ambiguous.

3.3 Properties and Inference Rules

The semantics defined above have some good properties:

Lemma 1. *New predicate functions can be safely appended to Λ, without changing the meaning of existing symbols in Λ. Formally, if $\Lambda' = (\Lambda, p(\overline{r}) \doteq \psi)$, where p is not defined in Λ, we have for every symbol q defined in Λ:*

$$\mathcal{J}_{\Lambda}(q) = \mathcal{J}_{\Lambda'}(q).$$

By this lemma, we can easily get:

Lemma 2. *Given a logic environment Λ:*
(1) we can safely append some new definitions to it, without changing semantics of symbols defined in Λ;
(2) if symbols $\overline{p}$ defined in Λ are not mentioned in other definitions in Λ, then we can safely remove them, without changing semantics of the other symbols defined in Λ. □

And by OOSL's semantics, it is straightforward to prove the following propositions:

Lemma 3. *Suppose $\sigma, O \models \psi$, we have:*
(1) if $\mathsf{dom}\,\sigma' \cap \mathsf{dom}\,\sigma = \emptyset$, then $\sigma \cup \sigma', O \models \psi$;
(2) if ψ does not contain variables in σ', then $\sigma - \sigma', O \models \psi$. Here $\sigma - \sigma'$ denotes $\{(x, r) \in \sigma \mid x \notin \mathsf{dom}\,\sigma'\}$. □

Lemma 4. $\sigma, O \models \psi[e/x]$, *if and only if $\sigma \oplus \{x \mapsto \sigma e\}, O \models \psi$.* □

Lemma 5. *Suppose $a_1, a_2, ..., a_k$ are all attributes of type T, then we have:*

$$\mathsf{obj}(r, T) \Leftrightarrow r.a_1 \mapsto - * r.a_2 \mapsto - * ... * r.a_k \mapsto - \qquad \square$$

Lemma 6. $\mathsf{obj}(r_1, \text{-}) * \mathsf{obj}(r_2, \text{-}) \Rightarrow r_1 \neq r_2.$ □

Lemma 7.

$$\begin{aligned}\mathbf{emp} * \psi &\Leftrightarrow \psi \\ \psi_1 * (\psi_1 \mathrel{-\!\!*} \psi_2) &\Leftrightarrow \psi_2 \\ \psi_1 \mathrel{-\!\!*} (\psi_2 \wedge \psi_3) &\Leftrightarrow (\psi_1 \mathrel{-\!\!*} \psi_2) \wedge (\psi_1 \mathrel{-\!\!*} \psi_3) \\ \psi_1 \mathrel{-\!\!*} \psi_2 \mathrel{-\!\!*} \psi_3 &\Leftrightarrow (\psi_1 * \psi_2) \mathrel{-\!\!*} \psi_3\end{aligned}$$

Proof. We prove the last statement. Note that $\psi_1 \mathrel{-\!\!*} \psi_2 \mathrel{-\!\!*} \psi_3$ is $\psi_1 \mathrel{-\!\!*} (\psi_2 \mathrel{-\!\!*} \psi_3)$.

$\Rightarrow$: Assume $\sigma, O \models \psi_1 \mathrel{-\!\!*} (\psi_2 \mathrel{-\!\!*} \psi_3)$. Take any O' such that $O' \perp O$ and $\sigma, O' \models \psi_1 * \psi_2$, by the definition of $*$, there exist O_1 and O_2 such that $O' = O_1 * O_2$, and $\sigma, O_1 \models \psi_1$, and $\sigma, O_2 \models \psi_2$. Because $O_1 \perp O_2 * O$ and the assumption, we know that $\sigma, O_1 * O \models \psi_2 \mathrel{-\!\!*} \psi_3$. From this fact, and $\sigma, O_2 \models \psi_2$ and $O_2 \perp O_1 * O$, we have $\sigma, O_1 * O_2 * O \models \psi_3$. This is exactly $\sigma, O' * O \models \psi_3$, thus we have the "$\Rightarrow$" proved.

$\Leftarrow$: Suppose $\sigma, O \models (\psi_1 * \psi_2) \mathrel{-\!\!*} \psi_3$. Take any O_1 such that $O_1 \perp O$ and $\sigma, O_1 \models \psi_1$, then take any O_2 such that $O_2 \perp O_1 * O$ and $\sigma, O_2 \models \psi_2$, now we need to prove that $\sigma, O_1 * O_2 * O \models \psi_3$. Because $O_1 * O_2 \perp O$ and $\sigma, O_1 * O_2 \models \psi_1 * \psi_2$, we have the result immediately. □

Many propositions in Separation Logic also hold in OOSL. For example, rules (axiom schemata) shown in the Section 3 of [15] are all valid here.

Intuitively, there are close connection between OOSL defined here and the Separation Logic. If we treat every tuple (r, a) as an address of memory cell, and define a suitable address transformation for the memory layout, then we may map the storage model of our logic to the storage model of Separation Logic. So, we conjecture that every proposition holding in Separation Logic, when it does not involve in address arithmetic, will hold in OO Separation Logic. We will investigate the relation between Separation Logic and OOSL in future.

Similar to Separation Logic, we can define the *pure*, *intuitionistic*, *strictly-exact* and *domain-exact* assertions. We find another important concept as follows.

Definition 7 (Separated Assertions). *Two assertions* ψ *and* ψ' *are* separated *from each other, iff for all stores* σ *and Opools* O, O', $\sigma, O \models \psi$ *and* $\sigma, O' \models \psi'$ *implies* $O \perp O'$. □

Lemma 8. $r_1.a \mapsto$ - *and* $r_2.b \mapsto$ - *are separated, provided that* $r_1 \neq r_2$*, or* a *and* b *are different attribute names.* □

For example, suppose we have a $Node$ class with fields $value$ and $next$. For a reference $r : Node$, we know $r.value \mapsto$ - and $r.next \mapsto$ - are separated. No corresponding concept is in original Separation Logic, due to the absence of attributes.

Lemma 9. *Suppose* ψ_1 *and* ψ_2 *are separated.* (1) *If* $\sigma, O_1 \models \psi_1$ *and* $\sigma, O_2 \models \psi_2$*, then* $\sigma, O_1 * O_2 \models \psi_1 * \psi_2$. (2) *If* $\sigma, O \models \psi_1 * \psi_2$*, there exists an unique partition of* $O = O_1 * O_2$*, that* $\sigma, O_1 \models \psi_1$ *and* $\sigma, O_2 \models \psi_2$. □

Lemma 10. ψ_1 *is separated from both* ψ_2 *and* ψ_3*, iff* ψ_1 *is separated from* $\psi_2 * \psi_3$. □

Theorem 1. *For any* ψ_1, ψ_2, ψ_3*, if* ψ_1 *and* ψ_2 *are separated from each other, then* $\psi_1 * (\psi_2 -\!\!* \psi_3) \Leftrightarrow \psi_2 -\!\!*(\psi_1 * \psi_3)$.

Proof. The proof is as follows:

$\Rightarrow$: For any σ and O such that $\sigma, O \models \psi_1 * (\psi_2 -\!\!* \psi_3)$, there exist O_1, O_2, such that $O_1 * O_2 = O$, $\sigma, O_1 \models \psi_1$, and $\sigma, O_2 \models \psi_2 -\!\!* \psi_3$. By the definition of $-\!\!*$, for any O_3 satisfying $O_2 \perp O_3$,

$$\sigma, O_3 \models \psi_2 \text{ implies } \sigma, O_2 * O_3 \models \psi_3.$$

Because ψ_1 and ψ_2 are separated, then by **Lemma 9**,

$$\sigma, O_3 \models \psi_2 \text{ implies } \sigma, O_1 * O_2 * O_3 \models \psi_1 * \psi_3.$$

This is $\sigma, O \models \psi_2 -\!\!*(\psi_1 * \psi_3)$.

$\Leftarrow$: For any σ and O that $\sigma, O \models \psi_2 -\!\!*(\psi_1 * \psi_3)$, for any O_1 that $O_1 \perp O$, if $\sigma, O_1 \models \psi_2$, then $\sigma, O_1 * O \models \psi_1 * \psi_3$. Now we fix this O_1. From $\sigma, O_1 * O \models \psi_1 * \psi_3$ we know there exist O_2 and O_3' such that $O_2 \perp O_3'$, $O_2 * O_3' = O_1 * O$, $\sigma, O_2 \models \psi_1$

and $\sigma, O'_3 \models \psi_3$. Because ψ_1, ψ_2 are separated, then $O_2 \perp O_1$. Thus $O'_3 = O_1 * O_3$ for some O_3. Now we have

$$\sigma, O_2 \models \psi_1,\ \sigma, O_1 \models \psi_2,\ \text{and}\ \sigma, O_1 * O_3 \models \psi_3.$$

Then we have $\sigma, O_3 \models \psi_2 -\!\!* \psi_3$, because the choice of O_1 needs no extra restriction. Thus $\sigma, O \models \psi_1 * (\psi_2 -\!\!* \psi_3)$, because $O = O_2 * O_3$. □

The concept of *separated assertions* is very useful in reasoning OO programs. Taking the $Node$ class above as an example, it allows us to combine relative attributes of a $Node$ object together:

$$\begin{aligned} & r_1.value \mapsto - * (r_2.value \mapsto - -\!\!* r_1.next \mapsto -) \\ \Leftrightarrow\ & r_2.value \mapsto - -\!\!*(r_1.value \mapsto - * r_1.next \mapsto -) \end{aligned}$$

3.4 Discussion

In this section, we discuss some expressiveness and extension issues about OOSL.

As presented above, we define a power assertion language for OOSL, especially the user-defined predicates, which notably enhance the expressiveness of OOSL. With OOSL, We can describe and infer recursive data structures, and some important properties between objects, such as accessibility, dangling and so on. Since our logic adopts classical semantics, it is more expressive than its intuitionistic cousin, e.g., what defined in [14]. We can use OOSL to describe the program state precisely, especially the Opool, i.e., what is in or is not in an Opool.

On the other hand, the primitive assertions of OOSL are very simple and specific, so we cannot describe quantitative relation or more complicated mathematical concepts with OOSL. But it is not difficult to extend OOSL to support these concepts. For example, if we want to support integer arithmetic in OOSL, we should

- add primitive assertions about integer,
- expand user-defined predicates with integer arguments,
- expand quantifiers $\exists$ and $\forall$ to support integer,
- define semantics for new adding assertions.

After these modifications, we can describe and infer properties involving integers with OOSL. In fact, we can combine OOSL and other mathematical theories freely, such as theories about sequences and trees, if they are orthogonal.

4 A Simple OO Language and Its Inference Rules

In this section, we study a simple OO language. For simplicity, we only consider basic commands here. High-level features, i.e., concepts related to method and class, involving more static structure and type information, will be studied in our further works. We demand that our storage model and OOSL are ready to deal with them.

The syntax of the language is as follows:

$$\begin{aligned} e &::= \mathbf{true} \mid \mathbf{false} \mid \mathbf{null} \mid x \\ b &::= \mathbf{true} \mid \mathbf{false} \mid e = e \mid \neg b \mid b \wedge b \mid b \vee b \\ c &::= \mathbf{skip} \mid x := e \mid x.a := e \mid x_1 := x_2.a \mid \\ &\quad\;\; x := \mathbf{new}\ C() \mid c; c \mid \mathbf{if}\ b\ c\ \mathbf{else}\ c \mid \mathbf{while}\ b\ c \end{aligned}$$

where:

- x is a variable, C a class name or $\mathbf{Object}$, a an attribute name. We adopt restricted forms for expressions e, so their values depend only on the store. Complex expressions can be encoded with the help of auxiliary variables and assignments.
- Assignments are restricted to a number of special forms. Beside the plain assignment $x := e$, we have mutation assignment $x.a := e$, and lookup assignment $x_1 := x_2.a$. Other general cases can be also encoded by these forms. For instance, one can use $x := y.a$ and then refer to $x.a'$ as a replacement of $y.a.a'$.
- $x := \mathbf{new}\ C()$ creates a new raw object, that is this command do not initialize the new object.

We define Hoare-Triple like inference rules for the language. Specifications take the form $\{P\}\ c\ \{Q\}$, where P is the precondition, Q is the postcondition, and c is a command. By $\{P\}c\{Q\}$, we mean that whenever P holds before the execution of command c, then predicate Q holds after the termination of c.

We list basic rules in **Fig. 2**. Here we treat boolean expressions as OOSL assertions, because every boolean expression can be easily mapped to an assertion in OOSL.

Beside basic rules, we have *Frame Rule* that is essential for local reasoning [15].

$$\frac{\{P\}\ c\ \{Q\} \quad FV(R) \cap md(c) = \emptyset}{\{P * R\}\ c\ \{Q * R\}} \qquad \text{(FRAME)}$$

where $FV(R)$ is the set of all program variables in R, and $md(c)$ denotes the variable set modified by c with the following definition:

$$md(c) = \begin{cases} \{x\}, & \textit{if}\ \ c\ \ \textit{is}\ \ x := \ldots \\ md(c_1) \cup md(c_2), & \textit{if}\ \ c\ \ \textit{is}\ \ c_1; c_2 \\ md(c_1) \cup md(c_2), & \textit{if}\ \ c\ \ \textit{is}\ \ \mathbf{if}\ b\ c_1\ \mathbf{else}\ c_2 \\ md(c), & \textit{if}\ \ c\ \ \textit{is}\ \ \mathbf{while}\ b\ c \\ \emptyset, & \textit{otherwise} \end{cases}$$

In this paper, we only define the local rules. We can define global rules and backwards rules, as in [15]. For example, here is the backwards reasoning rule for mutation:

$$\{(v = r_1) \wedge (e = r_2) \wedge (r_1.a \mapsto - * (r_1.a \mapsto r_2 -\!\!* P))\}\ v.a = e\ \{P\} \quad \text{(ASN-II BACK)}$$

Based on Rule (CONS), (FRAME) and Lemma 7, it is easy to prove that this rule is equivalent to Rule (ASN-II).

We use a little example to end this section. This example illustrates that two newly created empty objects are different. As mentioned above, $\mathbf{Object}$ has no attributes, by

$$\{P\}\ \mathbf{skip}\ \{P\} \qquad \text{(SKIP)}$$

$$\{P[e/x]\}\ x := e\ \{P\} \qquad \text{(ASN-I)}$$

$$\{x = r_1 \wedge e = r_2 \wedge r_1.a \mapsto -\}\ x.a := e\ \{x = r_1 \wedge e = r_2 \wedge r_1.a \mapsto r_2\} \qquad \text{(ASN-II)}$$

$$\{x_2 = r_1 \wedge r_1.a \mapsto r_2\}\ x_1 := x_2.a\ \{x_1 = r_2 \wedge x_2 = r_1 \wedge r_1.a \mapsto r_2\} \qquad \text{(ASN-III)}$$

$$\{\mathbf{emp}\}\ x := \mathbf{new}\ C()\ \{\exists r \cdot x = r \wedge \mathsf{obj}(r, C)\} \qquad \text{(NEW)}$$

$$\frac{\{P\}\ c_1\ \{Q\}, \quad \{Q\}\ c_2\ \{R\}}{\{P\}\ c_1; c_2\ \{R\}} \qquad \text{(SEQ)}$$

$$\frac{\{b \wedge P\}\ c_1\ \{Q\}, \quad \{\neg b \wedge P\}\ c_2\ \{Q\}}{\{P\}\ \mathbf{if}\ \ b\, c_1\ \mathbf{else}\ \ c_2\ \{Q\}} \qquad \text{(COND)}$$

$$\frac{\{b \wedge I\}\ c\ \{I\}}{\{I\}\ \mathbf{while}\ b\ c\ \{\neg b \wedge I\}} \qquad \text{(ITER)}$$

$$\frac{P \Rightarrow P', \quad \{P'\}\ c\ \{Q'\}, \quad Q' \Rightarrow Q}{\{P\}\ c\ \{Q\}} \qquad \text{(CONS)}$$

$$\frac{\{P\}\ c\ \{Q\}}{\{\exists r \cdot P\}\ c\ \{\exists r \cdot Q\}} \quad r \text{ is free in } P \text{ and } Q \qquad \text{(EX)}$$

Fig. 2. Inference Rules for the OO language

Rule (NEW), (FRAME) and Lemma 6,7, we have the following deduction:

$$\begin{array}{l}
\{\mathbf{true}\} \\
\{\mathbf{emp} * \mathbf{true}\} \\
x\ :=\ \mathbf{new\ Object}(); \\
\{\exists r \cdot x = r \wedge \mathsf{obj}(r, \mathbf{Object}) * \mathbf{true}\} \\
y\ :=\ \mathbf{new\ Object}(); \\
\{\exists r_1, r_2 \cdot x = r_1 \wedge y = r_2 \wedge \mathsf{obj}(r_1, \mathbf{Object}) * \mathsf{obj}(r_2, \mathbf{Object}) * \mathbf{true}\} \\
\{x \neq y\}
\end{array}$$

This example shows that OOSL's accurate treatment for empty objects.

5 A Case Study

In this section, we take Schorr-Waite Marking (SWM) Algorithm as an example to show how the specification and verification can be carried on with our logic and inference rules. **Fig. 3** gives an implementation of SWM algorithm in our language. Class $Node$ is the graph node class which has four attributes: $left$ and $right$ are links to the left and right subnodes respectively, flag $mark$ indicates if the node is marked, and flag $check$ is used internally to indicate if its left part has been visited.

To verify the program, we must specify that given any unmarked graph pointed by $root$, after the execution $schorr\_waite$, all nodes in the graph are marked and the graph structure is preserved. Complete verification for these two properties is complicated, especially for the second property, for which we must introduce some mathematical concepts for graphs. Yang [16] presented the first work on verifying SWM with Separation

```
class $Node$ {
  $Node\ left,\ right;$
  Bool $mark,\ check;$ $/*\ whether\ left\ subtree\ has\ been\ visited\ */$
}

void $schorr\_waite(Node\ root)$ {
  $Node\ t,\ p,\ q,\ s;$
  $t := root;\ p :=$ null;
  while $(p \neq$ null $\vee\ (t \neq$ null $\wedge \neg t.mark))$ {
    if $(t =$ null $\vee\ t.mark)$ {
      if $(p.check)$ { $/*\ pop\ */$
        $q := t;\ t := p;\ p := p.right;\ t.right := q;$
      }
      else { $/*\ swing\ */$
        $q := t;\ t := p.right;\ s := p.left;\ p.right := s;\ p.left := q;\ p.check :=$ true;
      }
    }
    else { $/*\ push\ */$
      $q := p;\ p := t;\ t := t.left;\ p.left := q;\ p.mark :=$ true; $p.check :=$ false;
    }
  }
}
```

Fig. 3. Implementation of Schorr-Waite Marking Algorithm

$$
\begin{array}{ll}
\mathsf{node}(r,r_1,r_2,c,m) & \doteq r.left \mapsto r_1 * r.right \mapsto r_2 * r.check \mapsto c * r.mark \mapsto m \\
\mathsf{mtree}(r) & \doteq (r = \mathsf{rnull} \wedge \mathbf{emp}) \vee \\
& \quad (\exists r_1, r_2 \cdot \mathsf{node}(r,r_1,r_2,-,\mathsf{rtrue}) * \mathsf{mtree}(r_1) * \mathsf{mtree}(r_2)) \\
\mathsf{utree}(r) & \doteq (r = \mathsf{rnull} \wedge \mathbf{emp}) \vee \\
& \quad (\exists r_1, r_2 \cdot \mathsf{node}(r,r_1,r_2,-,\mathsf{rfalse}) * \mathsf{utree}(r_1) * \mathsf{utree}(r_2)) \\
\mathsf{sbot}(r) & \doteq \exists r_1, r_2, c \cdot \mathsf{node}(r_b,r_1,r_2,c,\mathsf{rtrue}) * \\
& \qquad ((c = \mathsf{rtrue} \wedge r_2 = \mathsf{rnull} \wedge \mathsf{mtree}(r_1)) \vee \\
& \qquad\; (c = \mathsf{rfalse} \wedge r_1 = \mathsf{rnull} \wedge \mathsf{utree}(r_2))) \\
\mathsf{sseg}(r_t, r_b) & \doteq (r_t = r_b \wedge \mathsf{sbot}(r_b)) \vee (\exists r_1, r_2, c \cdot \mathsf{node}(r,r_1,r_2,c,\mathsf{rtrue}) * \\
& \qquad ((c = \mathsf{rtrue} \wedge \mathsf{mtree}(r_1) * \mathsf{sseg}(r_2,r_b)) \vee \\
& \qquad\; (c = \mathsf{rfalse} \wedge \mathsf{utree}(r_2) * \mathsf{sseg}(r_1,r_b)))))
\end{array}
$$

Fig. 4. User-defined assertions for Schorr-Waite Algorithm

Logic, where he gave a complete verification of SWM on binary tree. For the verification, he introduced some auxiliary mathematical concepts, including tree and list. As an illustration possibly been included in the paper, here, we do not focus on a complete verification. We simplify the specification in two aspects: We require the input is a tree, and we do not care about the tree structure preservation. So we take a specification here as: given any unmarked tree, after the execution of the program, all nodes in the tree are marked. Though this specification is not complete, it is a good example to illustrate the usefulness and power of OOSL.

At first, we introduce some user-defined assertions, as shown in **Fig. 4**. Assertion node specifies a single tree node; $\mathsf{mtree}(r)$ and $\mathsf{utree}(r)$ specify that the whole tree from r is marked or unmarked, respectively. Assertions sbot and sseg talk about the implicit stack and the segment of nodes reachable through the stack. In details, $\mathsf{sbot}(r)$ specifies that r is the only node in the stack and has been marked; and if the flag $check$ of this node is true, then its left subtree is marked and its right subtree is null, otherwise, its left subtree is null and right subtree is unmarked. On the other hand, $\mathsf{sseg}(r_t, r_b)$ specifies a stack with r_t as its top element and r_b its bottom element. Further, if $r_t = r_b$, then the stack has only one node r_b; otherwise, every node in the stack has been visited, and if the $check$ flag of a node is true, then its left subtree is marked and its $right$ field records the next node in the stack, otherwise its right subtree is unmarked and its $left$ field records the next node in the stack.

Now we give a specification for the SWM program:

$$\{root = r_{root} \wedge \mathsf{utree}(r_{root})\}\ SWM\ \{root = r_{root} \wedge \mathsf{mtree}(r_{root})\} \qquad (1)$$

Here SWM represents the body of the function $schorr\_waite$ shown in **Fig. 3**.

For proving the specification, the key-point is defining a suitable loop invariant. We define the loop invariant I as follows (with auxiliaries Inv_p and Inv_r):

$$\begin{aligned}
I \quad &\doteq \exists r_t, r_p \cdot t = r_t \wedge p = r_p \wedge (r_p = \mathsf{rnull} \Rightarrow r_t = r_{root}) \wedge I_p(r_p) * I_t(r_t)\\
I_p(r_p) &\doteq (r_p = \mathsf{rnull} \wedge \mathbf{emp}) \vee (r_p \neq \mathsf{rnull} \wedge \mathsf{sseg}(r_p, r_{root}))\\
I_t(r_t) &\doteq \mathsf{mtree}(r_t) \vee \mathsf{utree}(r_t)
\end{aligned}$$

This loop invariant says:

- If p is null, which means the stack is empty, then the value of t must be $root$;
- The whole Opool consists of two separated parts. The first part, that is specified by $I_p(r_p)$, is the part of nodes reachable from the implicit stack with p referring to its top element. If p is null then this part is empty. The second part, that is specified by $I_p(r_t)$, is a tree denoted by t. The nodes in the tree must be all marked or unmarked.

We can simply prove the following facts:

The precondition establishes the loop invariant:

$$\begin{aligned}
&\{root = r_{root} \wedge \mathsf{utree}(r_{root})\}\\
&t := root;\ p := \mathbf{null};\\
&\{root = r_{root} \wedge t = r_{root} \wedge p = \mathsf{rnull} \wedge \mathsf{utree}(r_{root})\}\\
&\{I\}
\end{aligned}$$

The postcondition holds when the loop ends:

$$\begin{aligned}
&(\exists r_p, r_t \cdot p{=}r_p \wedge t{=}r_t \wedge r_p{=}\mathsf{rnull} \wedge (r_t = \mathsf{rnull} \vee r_t.mark \hookrightarrow \mathsf{rtrue})) \wedge I\\
\Rightarrow\ & t = r_{root} \wedge \mathsf{mtree}(r_{root})
\end{aligned}$$

Now we prove that the loop invariant is preserved by the loop body. The whole proof is split into three cases according to the conditional branches, and all necessary lemmas and rules used in the deduction can be found in Section 3.3 and 4.

The following is the verification for case *Pop*. The other two cases are similar and we omit them here.

Case *Pop* : The condition is $p \neq \textbf{null} \wedge (t = \textbf{null} \vee t.mark) \wedge p.check$. We have the following deduction:

$$\begin{array}{l}
\{(\exists r_t, r_p \cdot t = r_t \wedge p = r_p \wedge r_p \neq \mathsf{rnull} \wedge \\
\quad (r_t = \mathsf{rnull} \vee r_t.mark \hookrightarrow \mathsf{rtrue}) \wedge r_p.check \hookrightarrow \mathsf{rtrue}) \wedge I\} \\
\{\exists r_t, r_p, r_{pl}, r_{pr} \cdot t = r_t \wedge p = r_p \wedge \\
\quad (\mathsf{mtree}(r_t) * \\
\quad\; ((r_p = r_{root} \wedge r_{pr} = \mathsf{rnull} \wedge \mathsf{node}(r_p, r_{pl}, r_{pr}, \mathsf{rtrue}, \mathsf{rtrue}) * \mathsf{mtree}(r_{pl})) \vee \\
\quad\;\; (\mathsf{node}(r_p, r_{pl}, r_{pr}, \mathsf{rtrue}, \mathsf{rtrue}) * \mathsf{mtree}(r_{pl}) * \mathsf{sseg}(r_{pr}, r_{root}))))\} \\
\{\exists r_t, r_p, r_{pl}, r_{pr} \cdot t = r_t \wedge p = r_p \wedge \\
\quad (\mathsf{mtree}(r_t) * \mathsf{node}(r_p, r_{pl}, r_{pr}, \mathsf{rtrue}, \mathsf{rtrue}) * \\
\quad\; \mathsf{mtree}(r_{pl}) * ((r_p = r_{root} \wedge r_{pr} = \mathsf{rnull} \wedge \textbf{emp}) \vee \mathsf{sseg}(r_{pr}, r_{root}))))\} \\
q := t;\ t := p;\ p := p.right; \\
\{\exists r_t, r_p, r_{pl}, r_{pr} \cdot q = r_t \wedge t = r_p \wedge p = r_{pr} \wedge \\
\quad (\mathsf{mtree}(r_t) * \mathsf{node}(r_p, r_{pl}, r_{pr}, \mathsf{rtrue}, \mathsf{rtrue}) * \mathsf{mtree}(r_{pl}) * \\
\quad\; ((r_p = r_{root} \wedge r_{pr} = \mathsf{rnull} \wedge \textbf{emp}) \vee \mathsf{sseg}(r_{pr}, r_{root})))\} \\
t.right := q; \\
\{\exists r_t, r_p, r_{pl}, r_{pr} \cdot q = r_t \wedge t = r_p \wedge p = r_{pr} \wedge \\
\quad (\mathsf{mtree}(r_t) * \mathsf{node}(r_p, r_{pl}, r_t, \mathsf{rtrue}, \mathsf{rtrue}) * \mathsf{mtree}(r_{pl}) * \\
\quad\; ((r_p = r_{root} \wedge r_{pr} = \mathsf{rnull} \wedge \textbf{emp}) \vee \mathsf{sseg}(r_{pr}, r_{root})))\} \\
\{\exists r_p, r_{pr} \cdot t = r_p \wedge p = r_{pr} \wedge (r_{pr} = \mathsf{rnull} \Rightarrow r_p = r_{root}) \wedge \\
\quad (\mathsf{mtree}(r_p) * ((r_{pr} = \mathsf{rnull} \wedge \textbf{emp}) \vee \mathsf{sseg}(r_{pr}, r_{root})))\} \\
\{I\}
\end{array}$$

With the proofs for the other two cases together, we conclude that the specification (1) holds. The proof for the full functional specification of Schorr-Waite Marking Algorithm will be one of our future works.

6 Related Work and Conclusion

To develop a full-armed logic framework for the specification and verification of OO programs is a long standing research goal in the software area. The work presented here is an attempt in this direction. As the last part of the paper, we overview some closely related work, make some comparisons, and list some future works.

Middelkoop tried to extend Separation Logic to OO domain in [9], where only the storage model is revised, and the assertion language remains. In their work, the separation conjunction operator $*$ is defined on the object level, but not on the attribute level; hence an object cannot be split. In this case, the atomic unit is a whole object, that limits the power of Frame Rule considerably.

Parkinson developed a revised Separation Logic for OO programs in his thesis [14] and some other papers. Although the start points are similar to ours, the framework is very different. In Parkinson's work, the program states are defined as:

$$\begin{array}{lll}
\text{Heaps} & \widehat{=} & (\text{OIDS} \times \text{FieldNames} \rightharpoonup_{\mathsf{fin}} \text{Values}) \times (\text{OIDS} \rightharpoonup_{\mathsf{fin}} \text{Class}) \\
\text{Stacks} & \widehat{=} & \text{Vars} \to \text{Values} \qquad \text{Interpretations} \ \widehat{=}\ \text{AuxVars} \to \text{Values} \\
\text{States} & \widehat{=} & \text{Heaps} \times \text{Stacks} \times \text{Interpretations} \qquad .
\end{array}$$

The first part of a heap $h \in \mathrm{Heaps}$ stores values of objects' attributes, and the second part stores their type information. An object is not explicit, but only a set of cells with the same id from OIDS. The additional component "$\mathrm{Interpretations}$" records values of logical variables. Taking this $\mathrm{Interpretations}$ into program states looks not nice, because it has no correspondent in practice. Clearly, logical variables are used only in verification, but not in execution. It is not nature to take them as a specific and independent part in program states. In this logic, operator $*$ separates only the first part of Heaps in states, thus different empty objects can not be separated. As seen, our state model is different, which records only information of program variables and objects. We have a novel definition for the separation of heaps (Opools), that makes it possible to separate the heaps efficiently even they contain empty objects.

Additionally, the logic in [14] adopts intuitionistic semantics, thus assertions preserve true with heap extension. This makes it impossible to express precise specifications about heaps, e.g., the simplest statement "current heap is empty". Consequently, no precise property about OO programs can be proved in this framework. Our logic takes the classical semantics, thus is more expressive [5]. The precise assertions are default, and intuitionistic assertions can also be written (ref. to [15]).

Parkinson *et al.* [13] developed a framework and some techniques based on their logic, for verifying OO programs modularly. We can also develop similar framework within our OOSL, which is our current work.

From these analysis and precognition, we make our choices. We take the reference model for OO languages, the assertion language based on Separation Logic, and the logic with classical semantics. Of course, what reported here is only a preliminary work.

In this paper, we present a state model for OO programs, and a novel definition for the separation of object heaps. Based on the storage model, we define an OO Separation Logic with some new assertion forms. We give a full treatment on user-defined basic assertions and introduce the concept of logic environment into our framework. We list the necessary conditions which guarantee the existence of the fixed point for a logic environment. We define semantics for the logic and prove some properties (reasoning rules) for it. We introduce a simple OO language with a set of inference rules based on the logic. The Schorr-Waite Marking algorithm is used as the example to illustrate how the specification and verification can be done here.

As for the future work, first, it would be interesting to study properties of OO Separation Logic, provide and prove more inference rules, in order to pave the way for more effective reasoning for OO programs. We also take interests in the connection between OOSL and the Separation Logic, as mentioned in Section 3.2.

Second, it is important to extend the language used here to support all higher level OO features, e.g., class declaration, method definition and invocation, inheritance, etc. Further, we should try to develop more powerful framework to do modular specification and verification like techniques in [11,13].

Third, accounting to the procedural paradigm, Weakest Precondition (WP) semantics plays a central role in semantics studies, and the foundation stone for many theoretical work deeply related to the software engineering, including specification, verification, refinement, programming from specifications [10], specification-based code generation, etc. A well-defined WP semantics might play similar role in OO paradigm. We will try to develop a WP Semantics for an OO language with all important OO features, which could enables us to define data refinement and program refinement. With WP semantics as a base, we could study program transformation, and the refinement relationship between programs/specifications at different abstract levels, therefore provide the possibility of programming from specifications or code generation.

References

1. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# programming system: An overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 49–69. Springer, Heidelberg (2005)
2. Bozga, M., Losif, R., Lakhnech, Y.: On logics of aliasing. In: Giacobazzi, R. (ed.) SAS 2004. LNCS, vol. 3148, pp. 344–360. Springer, Heidelberg (2004)
3. Chen, Y., Sanders, J.W.: A pointer logic for object diagrams. Technical report, International Institute for Software Technology, The United Nations University (2007)
4. Hoare, C.A.R., He, J.: A trace model for pointers and objects. In: Liu, H. (ed.) ECOOP 1999. LNCS, vol. 1628, pp. 344–360. Springer, Heidelberg (1999)
5. Ishtiaq, S., O'Hearn, P.W.: BI as an assertion language for mutable data structures. In: POPL 2001. ACM, New York (2001)
6. Ke, W., Liu, Z., Wang, S., Zhao, L.: A graph-based operational semantics of OO programs. In: Breitman, K., Cavalcanti, A. (eds.) ICFEM 2009. LNCS, vol. 5885, pp. 347–366. Springer, Heidelberg (2009)
7. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: A behavioral interface specification language for Java. SIGSOFT Software Engineering Notes 31(3), 1–38 (2006)
8. Leavens, G.T., Leino, K.R.M., Müller, P.: Specification and verification challenges for sequential object-oriented programs. Formal Asp. Comput. 19(2), 159–189 (2007)
9. Middelkoop, R., Huizing, K., Kuiper, R.: A separation logic proof system for a class-based language. In: Proceedings of the Workshop on Logics for Resources, Processes and Programs, LRPP (2004)
10. Morgan, C.: Programming from Specifications. Prentice Hall, Englewood Cliffs (1998)
11. Müller, P.: Modular Specification and Verification of Object-Oriented Programs. LNCS, vol. 2262. Springer, Heidelberg (2002)
12. Noble, J., Biddle, R., Tempero, E., Potanin, A., Clarke, D.: Towards a model of encapsulation. Technical report, Elvis Software Design Research Group (2003)
13. Parkinson, M.J., Bierman, G.M.: Separation logic, abstraction and inheritance. In: Principles of Programming Languages (POPL 2008). ACM, New York (2008)
14. Parkinson, M.J.: Local reasoning for Java. PhD thesis, University of Cambridge (2005)
15. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS 2002. IEEE Computer Society, Los Alamitos (2002) (Invited paper)
16. Yang, H.: Local Reasoning for Stateful Programs. PhD thesis, University of Illinois at Urbana-Champaign (Technical Report UIUCDCS-R-2001-2227) (2001)
17. Zhao, L., Liu, X., Liu, Z., Qiu, Z.: Graph transformations for object-oriented refinement. Formal Aspects in Computing 21(1), 103–131 (2009)

Model Checking Adaptive Multilevel Service Compositions[⋆]

Sabina Rossi

Dipartimento di Informatica, Università Ca' Foscari, Venezia (Italy)
srossi@dsi.unive.it

Abstract. In this paper we present a logic-based technique for verifying both security and correctness properties of multilevel service compositions. We define modal μ-calculus formulae interpreted over service configurations. Our formulae characterize those compositions which satisfy a non-interference property and are compliant, i.e., are both deadlock and livelock free. Moreover, we use filters as prescriptions of behavior (coercions to prevent service misbehavior) and we devise a model checking algorithm for adaptive service compositions which automatically synthesizes an adapting filter.

1 Introduction

A Service Oriented Architecture (SOA) provides a software architectural style to integrate loosely specified and coupled services that communicate with each other. Simple Object Access Protocol (SOAP)-based Web services are becoming the most common implementation of SOA. They are designed to support interoperable service-to-service interactions over a network. In such a context of heterogeneous systems where each application is leveraging the services of other applications, a service-oriented analysis and design process plays a significant role.

In this paper we develop a method for verifying both security and correctness properties of multilevel service compositions based on the use of model-checking techniques [9]. We specify service compositions in terms of behavioral contracts [5,6,8,16] which provide abstract descriptions of system behaviors by means of terms of a process algebra. Multi-party service compositions are modeled as the parallel composition of contracts. We define modal μ-calculus [15] formulae, interpreted over service configurations, characterizing those compositions which satisfy a non-interference property and are compliant, i.e., are both deadlock and livelock free. A model checker (like, e.g., NCSU Concurrency Workbench) can then be used to simultaneously check non-interference and compliance.

The notion of *non-interference* [12,14,22] we consider here is based on an information flow security model with dynamic policies. It demands that public synchronizations are unchanged as confidential communications are varied. Security policies are used to specify the security requirements of service components. In order to capture the dynamic nature, heterogeneity and lack of knowledge which are intrinsic features

[⋆] Work partially supported by the MIUR Project IPODS "Interacting Processes in Open-ended Distributed Systems".

Table 1. Syntax

Type environments	$\Gamma ::= \emptyset \mid \Gamma, p : \varsigma$	$p \in \mathcal{P},\ \varsigma \in \Sigma$
Actions	$\varphi ::= \bar{a}@u \mid a@u$	$a \in \mathcal{A},\ u \in \mathcal{P} \cup \mathcal{V}$
Contracts	$\sigma ::= \mathbf{1} \mid \mathbf{x} \mid \varphi.\sigma \mid \sigma + \sigma \mid \sigma \lfloor_{\Gamma}\oplus_{\Gamma} \rfloor \sigma \mid \mathtt{rec}(\mathbf{x})\, \sigma$	
Compositions	$C ::= p[\sigma] \mid C \parallel C$	

of modern web services, we allow policies to be dynamically specified by the service participants. In our model, for example, customers may formulate their security requirements by dynamically assign types (that are security annotations) to individual service components. We also consider the property of *compliance* which is widely used in the context of SOA as a formal device to identify well-formed service compositions, those whose interactions are free of synchronization errors.

Finally, we develop an algorithm for verifying adaptive service compositions. This is based on the use of filters, introduced in [7], as prescriptions of behaviour (coercions to prevent service misbehaviour). Security and correctness properties for adaptive multilevel service compositions are ensured by the automatic synthesis of an adapting filter.

Plan of the paper. Section 2 introduces the calculus for multilevel service compositions. Section 3 formalizes the notion of non-interference. Section 4 presents characteristic modal μ-calculus formulae for non-interference. Section 5 introduces the notion of compliance and defines a modal μ-calculus formula characterizing it. Section 6 presents an algorithm for adaptive service compositions. Finally, Section 7 concludes the paper.

2 The Calculus

We represent service contracts as terms of a value-passing CCS-like [18] process calculus that includes recursion and operators for external and internal choice. Parallel composition arises in *contract compositions* that we define as the parallel composition of a set of principals executing contracts. We presuppose a denumerable set of action names $\mathcal{A}$, ranged over by a, b, c, a denumerable set of principal identities $\mathcal{P}$, ranged over by p, q, r, and a denumerable set of variables $\mathcal{V}$, ranged over by x, y. The actions represent the basic units of observable behavior of the underlying services, while the principal names specify the peers providing the services.

In order to specify multilevel service compositions, we assign security levels to principal identities and express both contracts and compositions as typed terms of our calculus. Formally, we assume a complete lattice $\langle \Sigma, \preceq \rangle$ of security annotations, ranged over by ς, ϱ, where $\top$ and $\bot$ represent the top and the bottom elements of the lattice. We denote by $\sqcup$ and $\sqcap$, respectively, the join and meet operators over Σ. Type environments are used to assign security levels to principals. A type environments Γ is a finite mapping from principals and variables to security annotations. We define $\Gamma_1 \sqcup \Gamma_2$ (resp., $\Gamma_1 \sqcap \Gamma_2$) the type environment Γ such that $\Gamma(p) = \Gamma_i(p)$ if $p \notin dom(\Gamma_1) \cap dom(\Gamma_2)$ and $p \in dom(\Gamma_i)$, and $\Gamma(p) = \Gamma_1(p) \sqcup \Gamma_2(p)$ (resp., $\Gamma_1(p) \sqcap \Gamma_2(p)$) otherwise.

Table 2. Example of a travel agency

$$S = C[\sigma_C] \parallel T[\sigma_T] \parallel A_1[\sigma_A] \parallel A_2[\sigma_A]$$

$$\begin{aligned}\sigma_C = {} & \overline{\texttt{Req}}@T.\texttt{Lst}@T.(\,\overline{\texttt{Close}}@T.\mathbf{1}\lfloor_{\emptyset}\oplus{}_{T:\mathrm{H}}\rfloor(\,\overline{\texttt{Buy1}}@T.\overline{\texttt{Pay}}@T.\\ & \texttt{Get}@A_1.\mathbf{1}\lfloor_{A_1:\mathrm{H}}\oplus{}_{A_2:\mathrm{H}}\rfloor\overline{\texttt{Buy2}}@T.\overline{\texttt{Pay}}@T.\texttt{Get}@A_2.\mathbf{1}\,)\,)\end{aligned}$$

$$\begin{aligned}\sigma_T = {} & \texttt{Req}@x.\overline{\texttt{Inq}}@A_1.\overline{\texttt{Inq}}@A_2.\texttt{Price}@A_1.\texttt{Price}@A_2.\overline{\texttt{Lst}}@x.(\,\texttt{Close}@x.\mathbf{1} + {}\\ & \texttt{Buy1}@x.\overline{\texttt{Ord}}@A_1.\texttt{Pay}@x.\overline{\texttt{Conf}}@A_1.\mathbf{1} + \texttt{Buy2}@x.\overline{\texttt{Ord}}@A_2.\texttt{Pay}@x.\overline{\texttt{Conf}}@A_2.\mathbf{1})\end{aligned}$$

$$\sigma_A = \texttt{Inq}@x.\overline{\texttt{Price}}@x.(\,\texttt{Ord}@x.\texttt{Conf}@x.\overline{\texttt{Get}}@y.\mathbf{1} + \mathbf{1}\,)$$

Syntax. The syntax of our calculus is presented in Table 1. Term $\mathbf{1}$ indicates a contract that has reached a successful state. The contract $\bar{a}@p.\sigma$ describes a service that sends a message on a to principal p and then behaves as σ; syntactically, the principal identity p may be a variable, but it must be a name when the prefix is ready to fire. Dually, the input prefix $a@u.\sigma$ waits for an input on a from a particular/any principal and then continues as σ. If u is a variable x, then the input form is a binder for x with scope σ: upon synchronization with a principal p, x gets uniformly substituted by p in σ. The contract $\sigma + \sigma'$ denotes an external choice, guided by the environment. The contract $\sigma\lfloor_{\Gamma}\oplus{}_{\Gamma'}\rfloor\sigma'$ represents the internal choice between σ in the type environment Γ and σ' in the type environment Γ' made irrespective of the structure of the interacting components; the internal choice operator we adopt in this paper allows us to model the fact that a principal may dynamically change (upgrade) the security level of his interactions with other service components through the specific type environment associated to each choice. Finally, $\texttt{rec}(\mathbf{x})\,\sigma$ makes it possible to express iteration in the contract language. As usual, we assume a standard contractivity condition for recursion, requiring that recursive variables be guarded by a prefix. Given a principal $p \in \mathcal{P}$, we say that a contract σ is *p-compatible* if for all $\bar{a}@q$ and $a@q$ occurring in σ, q is different from p. A composition $p_1[\sigma_1] \parallel \cdots \parallel p_n[\sigma_n]$ of principals must be *well-formed* [4] to constitute a legal composition, namely: (i) the principal identities p_i's must all be pairwise different, and (ii) each contract σ_i, executed by principal p_i, is p_i-compatible. If $C = p_1[\sigma_1] \parallel \cdots \parallel p_n[\sigma_n]$ is a legal composition, we say that C is a $\{p_1, \ldots, p_n\}$-composition (dually, that $\{p_1, \ldots, p_n\}$ are the underlying principals for C). Throughout, we assume that contracts are closed (they have no free variables) and that compositions are well formed. Also, we often omit trailing $\mathbf{1}$'s.

A service component may modify the security level of its interactions with other components by assigning different security levels to the principals with which it is going to interact. However, it is reasonable to assume that a service component cannot downgrade the security level of other principals; moreover it cannot upgrade the level of its interactions with other components above its own level. These are the only typing constraints we assume. Such a typing discipline ensures that information does not explicitly flow from high to low, but it does not deal with implicit flows. Instead, we characterize noninterference in terms of the actions that typed service compositions may perform.

Table 3. Type system

$$\coprod(\mathbf{1}) = \emptyset \qquad \coprod(\mathbf{x}) = \emptyset \qquad \coprod(\varphi.\sigma) = \coprod(\sigma) \qquad \coprod(\sigma_1 + \sigma_2) = \coprod(\sigma_1) \sqcup \coprod(\sigma_2)$$

$$\coprod(\sigma_1 \lfloor_{\Gamma_1} \oplus {}_{\Gamma_2} \rfloor \sigma_2) = \Gamma_1 \sqcup \Gamma_2 \sqcup \coprod(\sigma_1) \sqcup \coprod(\sigma_2) \qquad \coprod(\mathtt{rec}(\mathbf{x})\,\sigma) = \coprod(\sigma)$$

$$\frac{}{\Gamma, p : \varsigma \vdash p[\sigma]}\ \varsigma \in \Sigma,\ \textstyle\bigsqcup_{q \in dom(\coprod(\sigma))} \coprod(\sigma)(q) \preceq \varsigma \qquad \frac{\Gamma \vdash C_1 \quad \Gamma \vdash C_2}{\Gamma \vdash C_1 \parallel C_2}$$

Example 1. Table 2 shows an example of a service contract composition. Four services are involved: $C[\sigma_C]$, $T[\sigma_T]$ and $A_i[\sigma_A]$ representing a *customer*, a *travel agency*, and two *airline companies*, respectively. The elementary actions represent business activities that result in messages being sent or received. For example, the action $\overline{\mathtt{Req}}@T$ undertaken by the customer results in a message being sent to the travel agency. The customer sends a request to the travel agency which then inquires the airlines to get the prices for the selected route. Each airline responds and the travel agency sends to the customer the list of the best prices. The customer decides whether to close the communication with the travel agency or to buy from one of the airlines. In the latter case the customer decides to assign a high security level (H) to both the travel agency and the chosen airline company in order to safeguard the confidentiality of the purchasing data. The travel agency orders the ticket from the selected airline and takes a deposit (or a full payment) from the customer. As soon as the airline receives the confirmation of the payment, the ticket is issued to the customer. □

Type System. The typing rules reported in Table 3 ensure that, given a service composition with an underling set π of principals, every $p \in \pi$ cannot upgrade the security level of the principals in π (including p) above the level of p itself. The judgments take the form $\Gamma \vdash C$, where Γ is a type environment and C is a service composition. We denote by $\coprod$ the function that associates to each contract σ the join of all the Γ_i occurring as a parameter of an internal choice in σ. We say that a service component $p[\sigma]$ is well-typed in Γ if p will never upgrade the security level of other principals over its own level; this is obtained by requiring that for all $q \in dom(\coprod(\sigma))$, it holds that $\coprod(\sigma)(q) \preceq \varsigma$ where ς is the security level of p.

Semantics. We define the dynamics of typed service compositions in terms of labelled transition systems (and a success predicate), with rules reported in Table 4. In the table, and in the whole paper, λ ranges over visible contract typed actions $\bar{a}@p$, $a@p$ and silent actions τ; δ ranges over service composition actions $a_{p\to q}$, $\bar{a}_{p\to q}$ and τ. We say that $\Gamma \rhd C$ is a *configuration* if Γ is a type environment and C is a $\{p_1, \ldots, p_n\}$-service composition such that $\{p_1, \ldots, p_n\} \subseteq dom(\Gamma)$.

The first block of rules defines the successful states of a contract and a composition, which are those that expose the successful term $\mathbf{1}$ at top level, or immediately under an external choice (up-to recursive unfoldings). Notice that a composition is successful only when all its components are successful. The second block of rules defines the

Table 4. Typed contract and composition transitions

Contract and composition satisfaction: $\sigma \checkmark$

$$\mathbf{1}\checkmark \qquad \frac{\sigma_i \checkmark}{\sigma_1 + \sigma_2 \checkmark} \qquad \frac{\sigma\{\mathbf{x} := \mathtt{rec}(\mathbf{x})\,\sigma\} \checkmark}{\mathtt{rec}(\mathbf{x})\,\sigma \checkmark} \qquad \frac{\sigma \checkmark}{p[\sigma] \checkmark} \qquad \frac{C_1 \checkmark \quad C_2 \checkmark}{C_1 \parallel C_2 \checkmark}$$

Typed contract transitions: $\Gamma \rhd \sigma \xrightarrow{\lambda} \Gamma' \rhd \sigma'$

$$\Gamma \rhd a@p.\sigma \xrightarrow{a@p} \Gamma \rhd \sigma \qquad \Gamma \rhd a@x.\sigma \xrightarrow{a@p} \Gamma \rhd \sigma\{x := p\}$$

$$\Gamma \rhd \bar{a}@p.\sigma \xrightarrow{\bar{a}@p} \Gamma \rhd \sigma \qquad \Gamma \rhd \sigma_1 \lfloor_{\Gamma_1} \oplus {}_{\Gamma_2} \rfloor \sigma_2 \xrightarrow{\tau} \Gamma \sqcup \Gamma_i \rhd \sigma_i \quad (i = 1, 2)$$

$$\frac{\Gamma \rhd \sigma_i \xrightarrow{\lambda} \Gamma' \rhd \sigma}{\Gamma \rhd \sigma_1 + \sigma_2 \xrightarrow{\lambda} \Gamma' \rhd \sigma}\,(i = 1, 2) \qquad \frac{\Gamma \rhd \sigma\{\mathbf{x} := \mathtt{rec}(\mathbf{x})\,\sigma\} \xrightarrow{\lambda} \Gamma' \rhd \sigma'}{\Gamma \rhd \mathtt{rec}(\mathbf{x})\,\sigma \xrightarrow{\lambda} \Gamma' \rhd \sigma'}$$

Typed composition transitions: $\Gamma \rhd C \xrightarrow{\delta} \Gamma' \rhd C'$

$$\frac{\Gamma \rhd \sigma \xrightarrow{a@p} \Gamma \rhd \sigma'}{\Gamma \rhd q[\sigma] \xrightarrow{a_{p\to q}} \Gamma \rhd q[\sigma']}\ p \in dom(\Gamma),\ p \neq q \qquad \frac{\Gamma \rhd \sigma \xrightarrow{\bar{a}@p} \Gamma \rhd \sigma'}{\Gamma \rhd q[\sigma] \xrightarrow{\bar{a}_{q\to p}} \Gamma \rhd q[\sigma']}\ p \neq q$$

$$\frac{\Gamma \rhd C_1 \xrightarrow{a_{p\to q}} \Gamma \rhd C_1' \quad \Gamma \rhd C_2 \xrightarrow{\bar{a}_{p\to q}} \Gamma \rhd C_2'}{\Gamma \rhd C_1 \parallel C_2 \xrightarrow{\tau} \Gamma \rhd C_1' \parallel C_2'}$$

$$\frac{\Gamma \rhd \sigma \xrightarrow{\tau} \Gamma' \rhd \sigma'}{\Gamma \rhd p[\sigma] \xrightarrow{\tau} \Gamma' \rhd p[\sigma']} \qquad \frac{\Gamma \rhd C_1 \xrightarrow{\delta} \Gamma' \rhd C_1'}{\Gamma \rhd C_1 \parallel C_2 \xrightarrow{\delta} \Gamma' \rhd C_1' \parallel C_2}$$

typed transitions for contracts. The rule for the internal choice ensures that a service component cannot downgrade the security level of other principals. Each typed contract transition yields a corresponding transition for the principal hosting the contract. Transitions for configurations are relative to the underlying set $dom(\Gamma)$ of principals and are entirely standard.

We use the following shorthands. We write $\Longrightarrow$ for the reflexive and transitive closure of $\xrightarrow{\tau}$, and $\overset{\delta}{\Longrightarrow}$ for $\Longrightarrow\xrightarrow{\delta}\Longrightarrow$. For a sequence of actions $w = \delta_1 \ldots \delta_n$, we write $\overset{w}{\Longrightarrow}$ to note $\overset{\delta_1}{\Longrightarrow} \cdots \overset{\delta_n}{\Longrightarrow}$. A computation for a configuration $\Gamma \rhd C$, is a sequence $\Gamma \rhd C = \Gamma_0 \rhd C_0 \xrightarrow{\tau} \Gamma_1 \rhd C_1 \xrightarrow{\tau} \ldots$ of internal actions.

Lemma 1 (Subject reduction). *Let Γ be a type environment and C be a service composition such that $\Gamma \rhd C$ is a configuration. If $\Gamma \rhd C$ is well-formed and $\Gamma \rhd C \xrightarrow{\tau} \Gamma' \rhd C'$, then $\Gamma' \rhd C'$ is well formed.* □

Table 5. Typed internal composition transitions

Typed internal composition transitions: $\Gamma \rhd C \overset{\alpha}{\hookrightarrow} \Gamma' \rhd C'$

$$\frac{\Gamma \rhd \sigma \xrightarrow{\tau} \Gamma' \rhd \sigma'}{\Gamma \rhd p[\sigma] \overset{\tau}{\hookrightarrow} \Gamma' \rhd p[\sigma']} \qquad \frac{\Gamma \rhd C_1 \overset{\alpha}{\hookrightarrow} \Gamma' \rhd C_1'}{\Gamma \rhd C_1 \parallel C_2 \overset{\alpha}{\hookrightarrow} \Gamma' \rhd C_1' \parallel C_2}$$

$$\frac{\Gamma \rhd C_1 \xrightarrow{a_{p\to q}} \Gamma \rhd C_1' \quad \Gamma \rhd C_2 \xrightarrow{\bar{a}_{p\to q}} \Gamma \rhd C_2'}{\Gamma \rhd C_1 \parallel C_2 \overset{\{a\}_{p\to q}}{\hookrightarrow} \Gamma \rhd C_1' \parallel C_2'}$$

Example 2. Consider again the service composition S of Example 1. Let Σ contain two security annotations, $\mathtt{L}$ (*public*) and H (*confidential*), with $\mathtt{L} \preceq \mathtt{H}$. Let Γ be the type environment $C : \mathtt{H}$, $T : \mathtt{L}$, $A_1 : \mathtt{L}$, $A_2 : \mathtt{L}$. The service composition S is well-typed in Γ, i.e., $\Gamma \vdash S$. □

3 Non-interference

The concept of *noninterference* [14] has been introduced to formalize the absence of information flow in multilevel systems. In the context of service compositions it demands that public interactions between service components are unchanged as secret communications are varied or, more generally, that the low level behaviour of the service composition is independent from the behaviour of its high components. In this way clients are assured that the data transmitted over the air to a web server remains confidential; in other words, sensitive data cannot be intercepted and understood by eavesdroppers.

The notion of non-interference we are going to introduce is relative to the internal behaviour of service compositions, i.e., we are interested in observing the synchronizations between service components. We thus refine the semantics of compositions in order to help (i) to distinguish a local contract move from a synchronization, and (ii) to identify the principals involved in every synchronization. This is captured by the rules collected in Table 5, where we use the relation $\hookrightarrow$ to represent typed synchronizations between service components. The τ label now indicates an internal action to a service component, while synchronizations between different peers in a composition are represented through a label of the form $\{a\}_{p\to q}$ meaning that principals p and q synchronize on action a. We let α range over the labels $\{a\}_{p\to q}$ and τ. We denote by $\overset{\tau}{\hookrightarrow\!\!\!\!\to}$ a possible empty sequence of $\overset{\tau}{\hookrightarrow}$ and define $\overset{\{a\}_{p\to q}}{\hookrightarrow\!\!\!\!\to} \overset{def}{=} \overset{\tau}{\hookrightarrow\!\!\!\!\to} \overset{\{a\}_{p\to q}}{\hookrightarrow} \overset{\tau}{\hookrightarrow\!\!\!\!\to}$.

The two semantics for service compositions, one expressed in terms of $\longrightarrow$ and the other one expressed in terms of $\hookrightarrow$, are related as follows.

Lemma 2. *Let $\Gamma \rhd C$ be a service configuration.*

- $\Gamma \rhd C \overset{\tau}{\hookrightarrow} \Gamma' \rhd C'$ *if and only if* $C = C_1 \parallel p[\sigma] \parallel C_2$, $C' = C_1 \parallel p[\sigma'] \parallel C_2$ *and* $\sigma \xrightarrow{\tau} \sigma'$;

– $\Gamma \rhd C \xhookrightarrow{\{a\}_{p\to q}} \Gamma \rhd C'$ *if and only if* $C = C_1 \parallel p[\sigma] \parallel C_2 \parallel q[\rho] \parallel C_3$, $C' = C_1 \parallel p[\sigma'] \parallel C_2 \parallel q[\rho'] \parallel C_3$, $\sigma \overset{\bar{a}@q}{\rightarrowtail} \sigma'$ *and* $\rho \overset{a@p}{\rightarrowtail} \rho'$. □

In order to define our notion of non-interference, we need to be able to distinguish the component interactions at a given security clearance. As transitions are typed, we can assign a security level to them as follows: the level of a synchronization depends on the level of the principals performing it. More precisely, the level of the synchronization $\{a\}_{p\to q}$ in the type environment Γ is defined as as:

$$\Gamma(\{a\}_{p\to q}) = \Gamma(p) \sqcap \Gamma(q).$$

Thus a ς-level synchronization is performed by principals whose security clearance is higher or equal to ς.

Behavioural Observations. We define behavioural observations in terms of equivalences that are parametric with respect to the security level $\varsigma \in \Sigma$ of the behaviour we want to observe. Such equivalences are relations over configurations that equate service compositions exhibiting the same ς-level component interactions. They are defined as a variant of the notion of *weak bisimulation* [18], an observation equivalence which allows one to observe the nondeterministic structure of the LTSs and focuses only on the observable actions.

In the following, we write $\Gamma_1 =_\varsigma \Gamma_2$ whenever $\{p \in dom(\Gamma_1) |\ \Gamma_1(p) \preceq \varsigma\} = \{p \in dom(\Gamma_2) |\ \Gamma_2(p) \preceq \varsigma\}$.

Definition 1 (Weak bisimulation on ς-low actions). *Let* $\varsigma \in \Sigma$. *A* weak bisimulation on ς-low actions *is the largest symmetric relation* $\approx_\varsigma$ *over configurations such that whenever* $\Gamma_1 \rhd C_1 \approx_\varsigma \Gamma_2 \rhd C_2$ *with* $\Gamma_1 =_\varsigma \Gamma_2$

– *if* $\Gamma_1 \rhd C_1 \xhookrightarrow{\alpha} \Gamma_1' \rhd C_1'$ *with* $\alpha = \tau$ *or* $\Gamma_1(\alpha) \preceq \varsigma$, *then there exist* Γ_2' *and* C_2' *such that* $\Gamma_2 \rhd C_2 \overset{\alpha}{\hookrightarrow\!\!\!\rightarrow} \Gamma_2' \rhd C_2'$ *with* $\Gamma_1' \rhd C_1' \approx_\varsigma \Gamma_2' \rhd C_2'$ *and* $\Gamma_1' =_\varsigma \Gamma_2'$;
– *if* $\Gamma_1 \rhd C_1 \xhookrightarrow{\alpha} \Gamma_1' \rhd C_1'$ *with* $\alpha \neq \tau$ *and* $\Gamma(\alpha) \not\preceq \varsigma$, *then there exist* Γ_2' *and* C_2' *such that either* $\Gamma_2 \rhd C_2 \overset{\alpha}{\hookrightarrow\!\!\!\rightarrow} \Gamma_2' \rhd C_2'$ *or* $\Gamma_2 \rhd C_2 \overset{\tau}{\hookrightarrow\!\!\!\rightarrow} \Gamma_2' \rhd C_2'$ *with* $\Gamma_1' \rhd C_1' \approx_\varsigma \Gamma_2' \rhd C_2'$ *and* $\Gamma_1' =_\varsigma \Gamma_2'$.

We write $\Gamma \models C_1 \approx_\varsigma C_2$ *when* $\Gamma \rhd C_1 \approx_\varsigma \Gamma \rhd C_2$. □

The notion of non-interference is inspired by [13] and is expressed in terms of a restriction operator $\cdot|_\varsigma$ which allows one to represent a service composition prevented from performing internal synchronizations of a level higher than ς. The semantics of $C|_\varsigma$ is described by the following rule:

$$\frac{\Gamma \rhd C \xhookrightarrow{\alpha} \Gamma' \rhd C'}{\Gamma \rhd C|_\varsigma \xhookrightarrow{\alpha} \Gamma' \rhd C'|_\varsigma}\ \Gamma(\alpha) \preceq \varsigma$$

Definition 2 (Non-interference). *Let* $\varsigma \in \Sigma$ *and* $\Gamma \rhd C$ *be a configuration. We say that the service composition* C *satisfies the non-interference property with respect to the level* ς *in the type environment* Γ, *denoted* $C \in \mathcal{NI}_{\Gamma,\varsigma}$, *if*

$$\Gamma \models C \approx_\varsigma C|_\varsigma.$$

□

Table 6. Example of a financial consulting platform

$$M = C[\sigma_C] \parallel F_1[\sigma_F] \parallel F_2[\sigma_F] \parallel S[\sigma_S]$$

$$\begin{aligned}\sigma_C = {} & \overline{\texttt{Inq}}@F_1.\overline{\texttt{Inq}}@F_2.\texttt{Plan}@F_1.\texttt{Plan}@F_2.\\ & (\,\overline{\texttt{Agree}}@F_1.\overline{\texttt{Close}}@F_2.\mathbf{1}\lfloor_{F_1:\texttt{H}}\oplus{}_{F_2:\texttt{H}}\rfloor\overline{\texttt{Agree}}@F_2.\overline{\texttt{Close}}@F_1.\mathbf{1}\,)\,)\end{aligned}$$

$$\sigma_F = \texttt{Inq}@x.\overline{\texttt{LookUp}}@S.\texttt{Quote}@x.\overline{\texttt{Plan}}@C.(\,\texttt{Agree}@x.\mathbf{1} + \texttt{Close}@x.\mathbf{1}\,)$$

$$\sigma_S = \texttt{LookUp}@x.\overline{\texttt{Quote}}@x.\mathbf{1}$$

$$M' = C[\sigma'_C] \parallel F_1[\sigma_F] \parallel F_2[\sigma_F] \parallel S[\sigma_S]$$

$$\begin{aligned}\sigma'_C = {} & \overline{\texttt{Inq}}@F_1.\overline{\texttt{Inq}}@F_2.\texttt{Plan}@F_1.\texttt{Plan}@F_2.\\ & (\,\overline{\texttt{Close}}@F_2.\overline{\texttt{Agree}}@F_1.\mathbf{1}\lfloor_{F_1:\texttt{H}}\oplus{}_{F_2:\texttt{H}}\rfloor\overline{\texttt{Close}}@F_1.\overline{\texttt{Agree}}@F_2.\mathbf{1}\,)\,)\end{aligned}$$

Example 3. Consider again the service composition S of Example 1 in the type environment Γ of Example 2. The property $S \in \mathcal{NI}_{\Gamma,\texttt{L}}$ holds. □

Example 4. Consider the service composition depicted in Table 6: it consists of a *client* C, two *financial consulting services* F_1 and F_2 and a *stock quote service provider* S. The client inquires the financial services to get investment advices. The financial services consult the stock quote service provider in order to look up information on the financial quotes. Then the financial services send their investment recommendations to the client which may decide whether or not adhere to the investment plan proposed by one of the financial services and close the connection with the other one.

Let $\Sigma = \{\texttt{L}, \texttt{H}\}$ with $\texttt{L} \preceq \texttt{H}$ and Γ be the type environment $C : \texttt{H}$, $F_1 : \texttt{L}$, $F_2 : \texttt{L}$, $S : \texttt{L}$. In this case we have that $M \notin \mathcal{NI}_{\Gamma,\texttt{L}}$. Indeed, there is a direct causality between the high level actions $\{\texttt{Agree}\}_{C\to F_i}$ and the low level action $\{\texttt{Close}\}_{C\to F_j}$ with $i \neq j$, performed after the clients makes the choice. As a consequence, if the client decides to accept the proposal of F_1 then F_2 knows that the customer has agreed to proceed with investment recommendation of F_1 by just observing that the action $\{\texttt{Close}\}_{C\to F_2}$ has been performed. The service composition can be made secure by letting $\{\texttt{Close}\}_{C\to F_j}$ be executed independently from $\{\texttt{Agree}\}_{C\to F_i}$ as in the composition M' which is obtained from M by replacing the contract σ_C with σ'_C. □

4 Modal Formulae for Non-interference

In this section we present a method for verifying whether $\Gamma \models C \approx_\varsigma C|_\varsigma$ which consists in defining a modal μ-calculus formula $\phi^{\approx_\varsigma}(\Gamma \rhd C)$ such that $\Gamma' \rhd C'$ satisfies $\phi^{\approx_\varsigma}(\Gamma \rhd C)$ iff $\Gamma \rhd C \approx_\varsigma \Gamma' \rhd C'$. The proofs are in the spirit of [19].

Table 7. Semantics of modal mu-calculus

$$
\begin{aligned}
M_{\Gamma \rhd C}(\mathbf{true})(\rho) &= S_{\Gamma \rhd C}\\
M_{\Gamma \rhd C}(\mathbf{false})(\rho) &= \emptyset\\
M_{\Gamma \rhd C}(\phi_1 \wedge \phi_2)(\rho) &= M_{\Gamma \rhd C}(\phi_1)(\rho) \cap M_{\Gamma \rhd C}(\phi_2)(\rho)\\
M_{\Gamma \rhd C}(\phi_1 \vee \phi_2)(\rho) &= M_{\Gamma \rhd C}(\phi_1)(\rho) \cup M_{\Gamma \rhd C}(\phi_2)(\rho)\\
M_{\Gamma \rhd C}(\langle \alpha \rangle \phi)(\rho) &= \{\Gamma' \rhd C' \mid \exists\, \Gamma'' \rhd C'' :\ \Gamma' \rhd C' \overset{\alpha}{\hookrightarrow} \Gamma'' \rhd C''\\
&\qquad\qquad \wedge\ \Gamma'' \rhd C'' \in M_{\Gamma \rhd C}(\phi)(\rho)\}\\
M_{\Gamma \rhd C}([\alpha]\phi)(\rho) &= \{\Gamma' \rhd C' \mid \forall\, \Gamma'' \rhd C'' :\ \Gamma' \rhd C' \overset{\alpha}{\hookrightarrow} \Gamma'' \rhd C''\\
&\qquad\qquad \Rightarrow\ \Gamma'' \rhd C'' \in M_{\Gamma \rhd C}(\phi)(\rho)\}\\
M_{\Gamma \rhd C}(X)(\rho) &= \rho(X)\\
M_{\Gamma \rhd C}(\mu X.\phi)(\rho) &= \bigcap \{x \subseteq S_{\Gamma \rhd C} \mid M_{\Gamma \rhd C}(\phi)(\rho[X \mapsto x]) \subseteq x\}\\
M_{\Gamma \rhd C}(\nu X.\phi)(\rho) &= \bigcup \{x \subseteq S_{\Gamma \rhd C} \mid M_{\Gamma \rhd C}(\phi)(\rho[X \mapsto x]) \supseteq x\}
\end{aligned}
$$

The modal μ-calculus [15] is a small, yet expressive process logic. We consider modal μ-calculus formulae constructed according to the following grammar:

$$\phi ::= \mathbf{true} \mid \mathbf{false} \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \langle \alpha \rangle \phi \mid [\alpha]\phi \mid X \mid \mu X.\phi \mid \nu X.\phi$$

where X ranges over an infinite set of variables and α over the labels $\{a\}_{p \to q}$ and τ. The *fixpoint operators* μX and νX bind the variable X and we adopt the usual notion of closed formula. For a finite set M of formulae, we write $\bigwedge M$ and $\bigvee M$ for the conjunction and disjunction of the formulae in M, where $\bigwedge \emptyset = \mathbf{true}$ and $\bigvee \emptyset = \mathbf{false}$.

Modal μ-calculus formulae are interpreted over service configurations modelled by LTS's. Let $\Gamma \rhd C$ be a configuration and $LTS(\Gamma \rhd C) = (S_{\Gamma \rhd C}, Act, \hookrightarrow)$ where $S_{\Gamma \rhd C}$ is the set of states reachable from $\Gamma \rhd C$ through $\overset{\alpha}{\hookrightarrow}$ and $\alpha \in Act$ denotes the action $\{a\}_{p \to q}$ or τ. The subset of configurations in $S_{\Gamma \rhd C}$ that satisfy a formula ϕ, noted by $M_{\Gamma \rhd C}(\phi)(\rho)$, is inductively defined in Table 7. ρ is an *environment*, i.e., it is a partial mapping $\rho : Var \not\to 2^{S_{\Gamma \rhd C}}$ that interprets at least the free variables of ϕ by subsets of $S_{\Gamma \rhd C}$. For a set $x \subseteq S_{\Gamma \rhd C}$ and a variable X, we write $\rho[X \mapsto x]$ for the environment that maps X to x and $Y \neq X$ to $\rho(Y)$ if ρ is defined on Y.

Intuitively, $\mathbf{true}$ and $\mathbf{false}$ hold for all resp. no states and $\wedge$ and $\vee$ are interpreted by conjunction and disjunction, $\langle \alpha \rangle \phi$ holds for a configuration $\Gamma' \rhd C' \in S_{\Gamma \rhd C}$ if there exists $\Gamma'' \rhd C''$ reachable from $\Gamma' \rhd C'$ with action α and satisfing ϕ, and $[\alpha]\phi$ holds for $\Gamma' \rhd C'$ if all configurations $\Gamma'' \rhd C''$ reachable from $\Gamma' \rhd C'$ with action α satisfy ϕ. The interpretation of a variable X is as prescribed by the environment. The formula $\mu X.\phi$, called *least fixpoint formula*, is interpreted by the smallest subset x of $S_{\Gamma \rhd C}$ that recurs when ϕ is interpreted with the substitution of x for X. Similarly, $\nu X.\phi$, called *greatest fixpoint formula*, is interpreted by the largest such set. Existence of such sets follows from the well-known Knaster-Tarski fixpoint theorem. As the meaning of a closed formula ϕ does not depend on the environment, we sometimes write $M_{\Gamma \rhd C}(\phi)$ for $M_{\Gamma \rhd C}(\phi)(\rho)$ where ρ is an arbitrary environment.

The set of configurations *satisfying* a closed formula ϕ is defined as $\mathit{Conf}(\phi) = \{\Gamma \rhd C \mid \Gamma \rhd C \in M_{\Gamma \rhd C}(\phi)\}$. We also refer to (closed) *equation systems*:

$$\mathit{Eqn} : X_1 = \phi_1 \ldots X_n = \phi_n$$

where $X_1, \ldots, X_n$ are distinct variables and $\phi_1, \ldots, \phi_n$ are formulae having at most $X_1, \ldots, X_n$ as free variables.

An environment $\rho : \{X_1, \ldots, X_n\} \rightarrow 2^{S_{\Gamma \rhd C}}$ is a *solution* of an equation system *Eqn*, if $\rho(X_i) = M_{\Gamma \rhd C}(\phi_i)(\rho)$. The fact that solutions always exist, is again a consequence of the Knaster-Tarski fixpoint theorem. In fact the set of environments that are candidates for solutions, $\mathit{Env}_{\Gamma \rhd C} = \{\rho \mid \rho : \{X_1, \ldots, X_n\} \rightarrow 2^{S_{\Gamma \rhd C}}\}$, together with the lifting $\sqsubseteq$ of the inclusion order on $2^{S_{\Gamma \rhd C}}$, defined by

$$\rho \sqsubseteq \rho' \text{ iff } \rho(X_i) \subseteq \rho'(X_i) \text{ for } i \in [1..n]$$

forms a complete lattice. Now, we can define the *equation functional* $\mathit{Func}^{\mathit{Eqn}}_{\Gamma \rhd C} : \mathit{Env}_{\Gamma \rhd C} \rightarrow \mathit{Env}_{\Gamma \rhd C}$ by $\mathit{Func}^{\mathit{Eqn}}_{\Gamma \rhd C}(\rho)(X_i) = M_{\Gamma \rhd C}(\phi_i)(\rho)$ for $i \in [1..n]$, the fixpoints of which are just the solutions of *Eqn*. $\mathit{Func}^{\mathit{Eqn}}_{\Gamma \rhd C}$ is monotonic and there is the largest solution $\nu \mathit{Func}^{\mathit{Eqn}}_{\Gamma \rhd C}$ of *Eqn* (with respect to $\sqsubseteq$), which we denote by $M_{\Gamma \rhd C}(\mathit{Eqn})$. This definition interprets equation systems on the configurations reachable by a given initial configuration $\Gamma \rhd C$. We lift this to configurations by agreeing that a configuration satisfies an equation system *Eqn*, if its initial state is in the largest solution of the first equation. Thus the set of configurations satisfying the equation system *Eqn* is $\mathit{Conf}(\mathit{Eqn}) = \{\Gamma \rhd C \mid \Gamma \rhd C \in M_{\Gamma \rhd C}(\mathit{Eqn})(X_1)\}$.

The relation $\approx_\varsigma$ can be characterized as the greatest fixpoint $\nu \mathit{Func}_{\approx_\varsigma}$ of the monotonic functional $\mathit{Func}_{\approx_\varsigma}$ on the complete lattice of relations $\mathcal{R}$ over configurations ordered by set inclusion, where $(\Gamma_1 \rhd C_1, \Gamma_2 \rhd C_2) \in \mathit{Func}_{\approx_\varsigma}(\mathcal{R})$ if and only if points (1) and (2) of Definition 1 hold. Thus a relation $\mathcal{R}$ is a weak bisimulation on ς-low actions if and only if $\mathcal{R} \subseteq \mathit{Func}_{\approx_\varsigma}(\mathcal{R})$, i.e., $\mathcal{R}$ is a *post-fixpoint* of $\mathit{Func}_{\approx_\varsigma}$. By the Knaster-Tarski fixpoint theorem, $\nu \mathit{Func}_{\approx_\varsigma}$ is the union of all the post-fixpoints of $\mathit{Func}_{\approx_\varsigma}$, i.e., it is the largest weak bisimulation on ς-low actions. If we restrict to the complete lattice of relations $\mathcal{R} \subseteq S_{\Gamma_1 \rhd C_1} \times S_{\Gamma_2 \rhd C_2}$ we obtain a monotonic functional $\mathit{Func}^{(\Gamma_1 \rhd C_1, \Gamma_2 \rhd C_2)}_{\approx_\varsigma}$ whose greatest fixpoint is exactly $\nu \mathit{Func}_{\approx_\varsigma} \cap (S_{\Gamma_1 \rhd C_1} \times S_{\Gamma_2 \rhd C_2})$, and this is enough to determine if $\Gamma_1 \rhd C_1 \approx_\varsigma \Gamma_2 \rhd C_2$.

Let $\Gamma \rhd C$ be finite-state, $\Gamma_1 \rhd C_1, \ldots, \Gamma_n \rhd C_n$ its $|S_{\Gamma \rhd C}| = n$ states, and $\Gamma_1 \rhd C_1 = \Gamma \rhd C$ its initial state. To derive a formula characterizing $\Gamma \rhd C$ up to $\approx_\varsigma$ we construct a *characteristic equation system* [19]:

$$\mathit{Eqn}_{\approx_\varsigma} : X_{\Gamma_1 \rhd C_1} = \phi^{\approx_\varsigma}_{\Gamma_1 \rhd C_1}, \ldots, X_{\Gamma_n \rhd C_n} = \phi^{\approx_\varsigma}_{\Gamma_n \rhd C_n}$$

consisting of one equation for each service configuration $\Gamma_1 \rhd C_1, \ldots, \Gamma_n \rhd C_n \in S_{\Gamma \rhd C}$. We define the formulae $\phi^{\approx_\varsigma}_{\Gamma_i \rhd C_i}$ such that the largest solution $M_{\Gamma \rhd C}(\mathit{Eqn}_{\approx_\varsigma})$ of $\mathit{Eqn}_{\approx_\varsigma}$ associates the variables $X_{\Gamma_i \rhd C_i}$ just with the states $\Gamma'_i \rhd C'_i$ of $S_{\Gamma \rhd C}$ which are weakly bisimilar on ς-low actions to $\Gamma_i \rhd C_i$, i.e., such that $M_{\Gamma \rhd C}(\mathit{Eqn}_{\approx_\varsigma})(X_{\Gamma_i \rhd C_i}) = \{\Gamma'_i \rhd C'_i \in S_{\Gamma \rhd C} \mid \Gamma_i \rhd C_i \approx_\varsigma \Gamma'_i \rhd C'_i\}$. Theorem 1 shows the exact form of such formulae. First we define:

$$\langle\!\langle \alpha \rangle\!\rangle_{\Gamma,\varsigma} \phi \overset{\text{def}}{=} \begin{cases} \langle\!\langle \alpha \rangle\!\rangle \phi & \text{if } \Gamma(\alpha) \preceq \varsigma \text{ or } \alpha = \tau \\ \langle\!\langle \alpha \rangle\!\rangle \phi \vee \langle\!\langle \tau \rangle\!\rangle \phi & \text{if } \Gamma(\alpha) \not\preceq \varsigma \text{ and } \alpha \neq \tau \end{cases}$$

where $\langle\!\langle\tau\rangle\!\rangle\phi \overset{\text{def}}{=} \mu X.\phi \vee \langle\tau\rangle X$ and $\langle\!\langle\alpha\rangle\!\rangle\phi \overset{\text{def}}{=} \langle\!\langle\tau\rangle\!\rangle\langle\alpha\rangle\langle\!\langle\tau\rangle\!\rangle\phi$. Let $\overset{\alpha}{\hookrightarrow\!\!\!\!\twoheadrightarrow}_{\Gamma,\varsigma}$ note either $\overset{\alpha}{\hookrightarrow\!\!\!\!\twoheadrightarrow}$ or $\overset{\tau}{\hookrightarrow\!\!\!\!\twoheadrightarrow}$. Then $\langle\!\langle\alpha\rangle\!\rangle_{\Gamma,\varsigma}$, $\langle\!\langle\tau\rangle\!\rangle$ and $\langle\!\langle\alpha\rangle\!\rangle$ correspond to $\overset{\alpha}{\hookrightarrow\!\!\!\!\twoheadrightarrow}_{\Gamma,\varsigma}$, $\overset{\tau}{\hookrightarrow\!\!\!\!\twoheadrightarrow}$ and $\overset{\alpha}{\hookrightarrow\!\!\!\!\twoheadrightarrow}$, since

- $M_{\Gamma\rhd C}(\langle\!\langle\alpha\rangle\!\rangle_{\Gamma,\varsigma}\phi)(\rho) = \{\Gamma' \rhd C' \mid \exists\, \Gamma'' \rhd C'' : \Gamma' \rhd C' \overset{\alpha}{\hookrightarrow\!\!\!\!\twoheadrightarrow}_{\Gamma,\varsigma} \Gamma'' \rhd C'' \wedge \Gamma'' \rhd C'' \in M_{\Gamma\rhd C}(\phi)(\rho)\}$
- $M_{\Gamma\rhd C}(\langle\!\langle\tau\rangle\!\rangle\phi)(\rho) = \{\Gamma' \rhd C' \mid \exists\, \Gamma'' \rhd C'' : \Gamma' \rhd C' \overset{\tau}{\hookrightarrow\!\!\!\!\twoheadrightarrow} \Gamma'' \rhd C'' \wedge \Gamma'' \rhd C'' \in M_{\Gamma\rhd C}(\phi)(\rho)\}$
- $M_{\Gamma\rhd C}(\langle\!\langle\alpha\rangle\!\rangle\phi)(\rho) = \{\Gamma' \rhd C' \mid \exists \Gamma'' \rhd C'' : \Gamma' \rhd C' \overset{\alpha}{\hookrightarrow\!\!\!\!\twoheadrightarrow} \Gamma'' \rhd C'' \wedge \Gamma'' \rhd C'' \in M_{\Gamma\rhd C}(\phi)(\rho)\}$.

Theorem 1. *Let $\phi^{\approx_\varsigma}_{\Gamma_i\rhd C_i}$ be the formula*

$$\begin{aligned}&\bigwedge\{\bigwedge\{\langle\!\langle\alpha\rangle\!\rangle_{\Gamma,\varsigma} X_{\Gamma_i'\rhd C_i'} \mid \Gamma_i \rhd C_i \overset{\alpha}{\hookrightarrow} \Gamma_i' \rhd C_i'\}\}\\ &\wedge \bigwedge\{[\alpha]\bigvee\{X_{\Gamma_i'\rhd C_i'} \mid \Gamma_i \rhd C_i \overset{\alpha}{\hookrightarrow\!\!\!\!\twoheadrightarrow}_{\Gamma,\varsigma} \Gamma_i' \rhd C_i'\}\}.\end{aligned}$$

Then $M_{\Gamma\rhd C}(Eqn_{\approx_\varsigma})(X_{\Gamma_i\rhd C_i}) = \{\Gamma_i' \rhd C_i' \in S_{\Gamma\rhd C} \mid \Gamma_i \rhd C_i \approx_\varsigma \Gamma_i' \rhd C_i'\}$. □

Characteristic formulae, i.e., *single* formulae characterizing configurations can be constructed by applying simple semantics-preserving transformation rules on equation systems as described in [19]. These rules are similar to the ones used by A. Mader in [17] as a mean of solving Boolean equation systems (with alternation) by Gauss elimination. Hence, since for any equation system Eqn there is a formula ϕ such that $Conf(Eqn) = Conf(\phi)$, we obtain that:

Theorem 2. *For any finite-state configuration $\Gamma \rhd C$ there is a modal μ-calculus formula $\phi^{\approx_\varsigma}(\Gamma \rhd C)$ such that $Conf(\phi^{\approx_\varsigma}(\Gamma \rhd C)) = \{\Gamma' \rhd C' \in S_{\Gamma\rhd C} \mid \Gamma' \rhd C' \approx_\varsigma (\Gamma' \rhd C')|_\varsigma\}$, that is the set of all the states reachable from $\Gamma \rhd C$ and satisfying $\mathcal{NI}_{\Gamma,\varsigma}$* □

5 A Modal Formula for Compliance

In this paper we refer to the notion of compliance for contract service compositions studied in [3]. Intuitively, a composition of services is compliant if it is deadlock and livelock free, i.e., it does not get stuck nor does it get trapped into infinite loops with no exit states. This notion is independent from the security levels of the principals involved in the component synchronizations. Therefore we omit trailing type environments in the definitions below, and write, e.g., $C \Longrightarrow C'$ to denote a transition of the form $\Gamma \rhd C \Longrightarrow \Gamma' \rhd C'$ for some type environments Γ and Γ'.

Definition 3 (Compliance). *Let C be a contract service composition. We say that C is* compliant, *noted $C \downarrow$, if for every C' such that $C \Longrightarrow C'$ there exists C'' such that $C' \Longrightarrow C''$ and $C'' \checkmark$.* □

In other words, the notion of compliance ensures that at each intermediate step of the computation in a service composition, each component has a way to reach a successful state (either autonomously, or via synchronizations). This is enough to avoid both deadlocks and livelocks.

Example 5. Consider the service composition S of Example 1. It holds that $\Gamma \rhd S$ is both compliant, i.e., $S\!\downarrow$, and non interfering, i.e., $S \in \mathcal{NI}_{\Gamma,\mathrm{L}}$. □

The notion of compliance can be equivalently expressed in terms of $\xhookrightarrow{\alpha}$ where α denotes a synchronization $\{a\}_{p\to q}$ or τ. More precisely, let $\gamma = \alpha_1, \ldots, \alpha_n$. We denote by $\xhookrightarrow{\gamma}\!\!\!\!\rightarrow$ the sequence of transitions $\xhookrightarrow{\alpha_1}\!\!\!\!\rightarrow \xhookrightarrow{\alpha_2}\!\!\!\!\rightarrow \ldots \xhookrightarrow{\alpha_n}\!\!\!\!\rightarrow$. Again we write $C \xhookrightarrow{\gamma}\!\!\!\!\rightarrow C'$ to denote a derivation $\Gamma \rhd C \xhookrightarrow{\gamma}\!\!\!\!\rightarrow \Gamma' \rhd C'$ for some type environments Γ and Γ'.

Proposition 1. *Let C be a contract service composition. It holds that C is* compliant, *$C\!\downarrow$, if and only if every C' such that $C \xhookrightarrow{\gamma'}\!\!\!\!\rightarrow C'$ for some $\gamma' \in Act^*$ there exist C'' and $\gamma'' \in Act^*$ such that $C' \xhookrightarrow{\gamma''}\!\!\!\!\rightarrow C''$ and $C''\checkmark$.* □

The modal μ-calculus formula that characterizes compliance is defined as follows:

$$\phi \stackrel{\mathrm{def}}{=} \mu X.\left((\checkmark) \vee \bigvee_{\alpha \in Act} (\langle\alpha\rangle X)\right) \wedge \neg\mu X.\left(\bigvee_{\alpha \in Act} (\langle\alpha\rangle X)\right)$$

where

$$\phi^c \stackrel{\mathrm{def}}{=} \mu X.\left(\bigwedge_{\alpha \in Act} ([\alpha]X) \wedge \phi\right)$$

The sub-formula $\neg\mu X.\left(\bigvee_{\alpha\in Act}(\langle\alpha\rangle X)\right)$ will ensure that any configuration satisfying ϕ^c doesn't get trapped into infinite loops without chances to reach a successful state. The next theorem characterizes the set of service configurations satisfying ϕ^c. The proof is given in [2].

Theorem 3. *Consider the modal μ-calculus formula ϕ^c defined above. It holds that $Conf(\phi^c) = \{\Gamma \rhd C \mid C\!\downarrow$ and Γ is a type environment$\}$.* □

Corollary 1. *A composition C is compliant if and only if $\Gamma \rhd C \in Conf(\phi^c)$ for some type environment Γ.* □

As a consequence of Theorems 2 and 3 we have:

Corollary 2. *Let $\varsigma \in \Sigma$, $\Gamma \rhd C$ be a configuration and*

$$\Phi^{\varsigma}_{\Gamma\rhd C} \stackrel{\mathrm{def}}{=} \phi^{\approx_\varsigma}(\Gamma \rhd C) \;\wedge\; \phi^c.$$

It holds that $\Gamma \rhd C \in Conf(\Phi^{\varsigma}_{\Gamma\rhd C})$ if and only if both $C \in \mathcal{NI}_{\Gamma,\varsigma}$ and $C\!\downarrow$. □

Using this method we can for instance exploit the model checker NCSU Concurrency Workbench (see [10]) to check whether both $C \in \mathcal{NI}_{\Gamma,\varsigma}$ and $C\!\downarrow$.

6 An Adaptive Algorithm

The model checking technique is based on the idea that the state transition graph of a finite-state system defines a Kripke structure, and efficient algorithms can be given for checking if the state graph defines a model of a given specification expressed in an appropriate temporal logic. In the explicit state approach the Kripke structure is represented extensionally, using conventional data structures such as adjacency matrices and linked lists so that each state and transition is enumerated explicitly. Moreover, in the global calculation approach, given a structure M and formula ϕ, the model checking algorithms calculate $\phi^M = \{s : M, s \models \phi\}$ that is the set of all states in M satisfying ϕ. We show how such algorithms can be exploited to develop an adaptive model checking technique for service compositions which adapts, when it is possible, the composition under investigation in such a way that it satisfies both non-interference and compliance. We use the filters, introduced in [7] and revised in [3], as prescriptions of behaviour.

Filters. A filter is the specification of the legal flow of actions for an individual contract. The syntax is as follows, while the semantics is defined in Table 8.

$$f \in \mathcal{F} \ := \ \mathbf{0} \mid \delta.f \mid f \times f \mid f \otimes f \mid X \mid \texttt{rec}(\mathbf{X})\, f$$

Definition 4 (Filter pre-order). *The filter pre-order $f \leq g$ is the largest relation such that if $f \xmapsto{\delta} f_\delta$ then $g \xmapsto{\delta} g_\delta$ and $f_\delta \leq g_\delta$.* □

We note $(\mathcal{F}, \sqsubseteq)$ the partial order induced by $\leq$: by abuse of notation, we identify a filter f with its equivalence class $[f]_\sim$, where $\sim$ is the symmetric closure of $\leq$. The union

Table 8. Dynamics of Filtered contract service compositions

Transitions for filters

$$\delta.f \xmapsto{\delta} f \qquad \frac{f\{X := \texttt{rec}(\mathbf{X})\, f\} \xmapsto{\delta} f'}{\texttt{rec}(\mathbf{X})\, f \xmapsto{\delta} f'} \qquad \frac{f \xmapsto{\delta} f_\delta \qquad g \xmapsto{\delta} g_\delta}{f \otimes g \xmapsto{\delta} f_\delta \otimes g_\delta}$$

$$\frac{f \xmapsto{\delta} f_\delta \qquad g \xmapsto{\delta} g_\delta}{f \times g \xmapsto{\delta} f_\delta \times g_\delta} \qquad \frac{f \xmapsto{\delta} f_\delta \qquad g \not\xmapsto{\delta}}{f \times g \xmapsto{\delta} f_\delta} \qquad \frac{f \not\xmapsto{\delta} \qquad g \xmapsto{\delta} g_\delta}{f \times g \xmapsto{\delta} g_\delta}$$

Transitions for filtered peers

$$\frac{\Gamma \rhd p[\sigma] \xrightarrow{\delta} \Gamma \rhd p[\sigma'] \qquad f \xmapsto{\delta} f'}{\Gamma \rhd f(p[\sigma]) \xrightarrow{\delta} \Gamma \rhd f'(p[\sigma'])}$$

$$\frac{\Gamma \rhd p[\sigma] \xrightarrow{\tau} \Gamma' \rhd p[\sigma']}{\Gamma \rhd f(p[\sigma])) \xrightarrow{\tau} \Gamma' \rhd f(p[\sigma'])} \qquad \frac{\Gamma \rhd p[\sigma] \checkmark}{\Gamma \rhd f(p[\sigma]) \checkmark}$$

and intersection of filters represent the glb and lub operators for $(\mathcal{F}, \sqsubseteq)$. Furthermore, if we assume a finite alphabet A of actions, the set of filters $\mathcal{F}_A$ insisting on A forms a complete lattice with **0** as bottom and the identity filter $I_A \stackrel{def}{=} \mathtt{rec}(\mathbf{X}) \prod_{\delta \in A} \delta.X$ as top element.

The application $\Gamma \rhd f(p[\sigma])$ blocks any action from $\Gamma \rhd p[\sigma]$ that is not explicitly enabled by f. Filters may be composed to help shape a service composition. Given a set π of principals, a composite π-filter F is a finite map from the principals in π to filters: $\{p \to f[p] \mid p \in \pi\}$. A π-filter may be applied to a π-composition:

$$\Gamma \rhd F(p_1[\sigma_1] \parallel \cdots \parallel p_n[\sigma_n]) ::= \Gamma \rhd F[p_1](p_1[\sigma_1]) \parallel \cdots \parallel \Gamma \rhd F[p_n](p_n[\sigma_n])$$

When we write $\Gamma \rhd F(C)$ we tacitly assume that the underlying set of principals for both F and C is π. The operators of union and intersection, as well as the ordering on filters extends directly to composite filters, as expected. Namely, for F and G π-filters and for $\bullet \in \{\times, \otimes\}$:

$$\begin{array}{ccll} F \leq_\pi G & \text{iff} & F[p] \leq G[p] & \text{for all } p \in \pi \\ (F \bullet_\pi G)[p] & \stackrel{def}{=} & F[p] \bullet G[p] & \text{for all } p \in \pi \end{array}$$

We generalize the syntax of service compositions by allowing the term $\Gamma \rhd F(C)$ to account for the application of filters on the components of C. The dynamics of filtered service compositions derives directly by combining the transitions in Tables 4 and 8.

Relevance. Below we present an algorithm that given a configuration $\Gamma \rhd C$ infers a composite filter F that fixes $\Gamma \rhd C$, whenever such F exists. The algorithm is so structured as to guarantee two important properties on the inferred filter. On the one hand, the filter is as permissive as possible, in that it is the greatest (with respect to the pre-order $\leq$) among the filters that fix $\Gamma \rhd C$. On the other side, the inferred filter is *relevant*, i.e., minimal in size: for any computation state reached by the service configuration via a series of τ transitions (local moves or synchronizations), the filter only enables actions that may be attempted at that state (either directly, or via a local choice), by one of the components of the service configuration.

Definition 5 (Relevance). *Let π be a set of principals and $\mathcal{C}$ be a non-empty set of π-configurations. A filter f is p-*relevant *in $\mathcal{C}$, written $f \propto_p \mathcal{C}$, if whenever $f \xmapsto{\delta} \widehat{f}$ one has $\delta \in \{a_{\_\to p}, \bar{a}_{p \to \_}\}$ and there exists $\Gamma \rhd C \in \mathcal{C}$ such that $\Gamma \rhd C \xhookrightarrow{\alpha}\!\!\!\!\to$ with $\alpha \in \{\{a\}_{\_\to p}, \{a\}_{p \to \_}\}$ and $\widehat{f} \propto_p \{\Gamma' \rhd C' \mid \Gamma \rhd C \xhookrightarrow{\alpha}\!\!\!\!\to \Gamma' \rhd C'\}$.*
A composite π-filter F is relevant *for $\mathcal{C}$, written $F \propto \mathcal{C}$, if $F(p) \propto_p \mathcal{C}$ for all $p \in \pi$.*
A composite π-filter is relevant for a π-configuration $\Gamma \rhd C$ if $F \propto \{\Gamma \rhd C\}$. □

The Algorithm. We describe an algorithm that synthesizes the $\sqsubseteq$-greatest relevant filter that fixes $\Gamma \rhd C$, if it exists, when $\Gamma \rhd C$ does not satisfy $\Phi^{\varsigma}_{\Gamma \rhd C}$.

As discussed above, a global model checking algorithm applied to a configuration $\Gamma \rhd C$ and the modal formula $\Phi^{\varsigma}_{\Gamma \rhd C}$ calculates the set of states in the reduction graph (tracing the states reached by means of synchronizations or internal moves) of $\Gamma \rhd C$ satisfying $\Phi^{\varsigma}_{\Gamma \rhd C}$. This is the input of our algorithm. The reduction graph can be represented as a directed graph $G = (V, E)$ with labelled edges and vertices. The vertices in

Procedure. `PushLabels(G)`

```
Input: A reduction graph $G = (V, E)$
Output: The graph $G$ updated

$done$ := false;
while $\neg$ $done$ do
    $done$ := true;
    foreach $\mathbf{u} \in V$ do
        $succ$ := false;  $fail$ := false;
        if $Adj[\mathbf{u}, \tau] \neq \emptyset$ then
            if $\exists \mathbf{v} \in Adj[\mathbf{u}, \tau] : result[\mathbf{v}] = \text{FAIL}$ then
                $fail$ := true;
            else if $\exists \mathbf{v} \in Adj[\mathbf{u}, \tau] : result[\mathbf{v}] = \text{SUCC}$ then
                $succ$ := true;
        else if $\exists (\alpha, \mathbf{v}) \in Adj[\mathbf{u}] \wedge result[\mathbf{v}] = \text{SUCC} \wedge \neg Conflict(\alpha, \mathbf{u})$ then
            $succ$ := true;
        if $succ \wedge result[\mathbf{u}] \neq \text{SUCC}$ then
            $result[\mathbf{u}]$ := SUCC;  $done$ := false;
        else if $fail \wedge result[\mathbf{u}] \neq \text{FAIL}$ then
            $result[\mathbf{u}]$ := FAIL;  $done$ := false
```

V represent the reachable states of $\Gamma \rhd C$. With each $\mathbf{v} \in V$ we associate two fields: $state[\mathbf{v}]$ gives the computation state (i.e., the derivative $\Gamma' \rhd C'$ of the initial state $\Gamma \rhd C$) associated with $\mathbf{v}$; $result[\mathbf{v}]$ is a tag SUCC or FAIL depending on whether the corresponding configuration satisfies $\Phi^{\varsigma}_{\Gamma \rhd C}$ or not as calculated by the model checker. An edge in E is a triple $(\mathbf{u}, \mathbf{v})_\alpha$ representing the transition $state[\mathbf{u}] \xhookrightarrow{\alpha} state[\mathbf{v}]$. Reduction graphs may be stored in a adjacency list representation, so that the set of outgoing edges for each $\mathbf{u} \in V$ can be retrieved as $Adj[\mathbf{u}]$: thus $(\mathbf{u}, \mathbf{v})_\alpha \in E$ iff $(\alpha, \mathbf{v}) \in Adj[\mathbf{u}]$. We also write $Adj[\mathbf{u}, \alpha]$ for the set $\{\mathbf{v} \in V \mid (\mathbf{u}, \mathbf{v})_\alpha \in E\}$. Vertices with no outgoing edges are called leaves. We denote by $root[G]$ the vertex representing the initial state $\Gamma \rhd C$.

The first step consists in *re-labelling* the graph G calculated by the model-checker in such a way that the result label at each vertex $\mathbf{u}$ is set to FAIL if there exists at least one silent transition from $\mathbf{u}$ to a FAIL vertex; it is set to SUCC if either there are no silent transitions from $\mathbf{u}$ to a FAIL vertex and there exists a silent transition from $\mathbf{u}$ to a SUCC vertex or there exists one non-silent and non-conflicting transition from $\mathbf{u}$ to a SUCC vertex. The procedure iteratively examines all the vertices in the graph until it reaches a fixed point. This computation is accomplished by the `PushLabels` procedure and uses the following auxiliary definitions. Let $locs(\alpha)$ be $\{p, q\}$ in case $\alpha = \{a_{p \to q}\}$, and $\emptyset$ in case $\alpha = \tau$. Let $G = (V, E)$ be a reduction graph, and $\alpha = \{a_{p \to q}\}$.

- A path $\varpi = (\mathbf{u}, \mathbf{u}_1)_{\alpha_1}, \ldots, (\mathbf{u}_{n-1}, \mathbf{v})_{\alpha_n}$ from $\mathbf{u}$ to $\mathbf{v}$ in G is *α-free* if $locs(\alpha) \cap locs(\alpha_i) = \emptyset$ for all i's.
- A vertex $\mathbf{v}$ is a *α-free descendant* of $\mathbf{u}$ in G (dually, $\mathbf{u}$ is a *α-free ancestor* of $\mathbf{v}$) if there is a α-free path from $\mathbf{u}$ to $\mathbf{v}$.

Function. `SuccessGraph(`G`)`

```
Input: A reduction graph $G = (V, E)$
Output: $G' = (V', E')$ the success sub-graph of $G$
$V' := (result[root[G]] = \text{SUCC})\ ?\ \{root[G]\} : \emptyset$;  $E' := \emptyset$;  $done :=$ false;
while $\neg\, done$ do
    $done :=$ true;
    foreach $(\mathbf{u}, \mathbf{v})_\alpha \in E \setminus E'$ do
        if $\mathbf{u} \in V' \wedge result[\mathbf{v}] = \text{SUCC} \wedge \neg Conflict(\alpha, \mathbf{u})$ then
            $V' := V' \cup \{\mathbf{v}\}$;  $E' := E' \cup \{(\mathbf{u}, \mathbf{v})_\alpha\}$;
            $done :=$ false
return $G' = (V', E')$;
```

- A vertex $\mathbf{u}$ *yields a conflict on* α if $\mathbf{u}$ has two distinct α-free descendants $\mathbf{v}_1$ and $\mathbf{v}_2$ such that $(\mathbf{v}_1, \mathbf{w}_1)_\alpha$ and $(\mathbf{v}_2, \mathbf{w}_2)_\alpha \in E$ and $result[\mathbf{w}_1] \neq result[\mathbf{w}_2]$.
- A vertex $\mathbf{v}$ has a conflict on α in G, noted $Conflict_G(\alpha, \mathbf{v})$ if $\mathbf{v}$ has a α-free ancestor yielding a conflict on α.

Intuitively, our algorithm will prune G by banning all the 'bad' synchronizations, and by preserving all the 'good' synchronizations that lead to nodes satisfying both non-interference and compliance. Due to the presence of internal choices, the same synchronization can look good at one point, but actually be bad. The definition of conflict formally captures this notion of ambiguous synchronizations.

Lemma 3. *After the call to* `PushLabels(`G`)`*, the following conditions hold for every node* $\mathbf{u}$ *in* G*:* (i) $result[\mathbf{u}] = \text{FAIL}$ *iff either there exists no* $(\mathbf{u}, \mathbf{v})_\alpha \in E$ *such that* $result[\mathbf{v}] = \text{SUCC}$ *and* $\neg Conflict_G(\alpha, \mathbf{u})$ *or there exists* $(\mathbf{u}, \mathbf{v})_\tau \in E$ *such that* $result[\mathbf{v}] = \text{FAIL}$*;* (ii) $result[\mathbf{u}] = \text{SUCC}$ *iff there exists no* $(\mathbf{u}, \mathbf{v})_\tau \in E$ *such that* $result[\mathbf{v}] = \text{FAIL}$ *and there exists either* $(\mathbf{u}, \mathbf{v})_\tau \in E$ *such that* $result[\mathbf{v}] = \text{SUCC}$ *or* $(\mathbf{u}, \mathbf{v})_\alpha \in E$ *with* $\alpha \neq \tau$*,* $\neg Conflict_G(\alpha, \mathbf{u})$ *and* $result[\mathbf{v}] = \text{SUCC}$*.* □

We say that a path ϖ in G is *successful* if $result[\mathbf{u}] = \text{SUCC}$ for every node $\mathbf{u}$ in ϖ, otherwise ϖ is *unsuccessful*. A node $\mathbf{u}$ is *root-successful* if it is reachable from $root[G]$ via a successful path, otherwise it is *root-unsuccessful*. The next step of the algorithm computes the sub-graph of G that only includes the root-successful vertices. This computation is accomplished by the `SuccessGraph` function.

Lemma 4. *Let* $G' = (E', V')$ *be the graph generated by* `SuccessGraph(`G`)`*. Then* $\mathbf{u} \in V'$ *if and only if* $\mathbf{u}$ *is root-successful in* G*.* □

The final step of the algorithm synthesizes the filter out of the success graph, in case this is not empty. Let $G' =$ `SuccessGraph(`G`)`, π be the underlying set of principals, and $F^{Alg}[\Phi^{\varsigma}_{\Gamma \rhd C}] = \texttt{ExtractFilter}_\pi(root[G], \emptyset, G')$.

Theorem 4 (Soundness and maximality). *Let* $\Gamma \rhd C$ *be a* π*-composition. Then* $\Gamma \rhd F^{Alg}[\Phi^{\varsigma}_{\Gamma \rhd C}](C)$ *is such that both* $F^{Alg}[\Phi^{\varsigma}_{\Gamma \rhd C}](C) \in \mathcal{NI}_{\Gamma,\varsigma}$ *and* $F^{Alg}[\Phi^{\varsigma}_{\Gamma \rhd C}](C) \downarrow$*. Also, if a filter* F *fixes* $\Gamma \rhd C$ *and is relevant for* $\Gamma \rhd C$*, then* $F \leq F^{Alg}[\Phi^{\varsigma}_{\Gamma \rhd C}]$*.* □

Function. `ExtractFilter`$_\pi$(**u**, U, G)

```
Input: $G = (V, E)$ a success graph. $\mathbf{u} \in V, U \subseteq V$
Output: $F$, an $\pi$-composite filter
$F[p] := \mathbf{0}$ for all $p \in \pi$;
if $state[\mathbf{u}]$ ✓ then
    return $F$;
if $\mathbf{u} \in U$ then
    $rec[\mathbf{u}] :=$ true; return $(X_{\mathbf{u}}, \ldots, X_{\mathbf{u}})$;
foreach $(\alpha, \mathbf{v}) \in Adj[\mathbf{u}]$ do
    $F_{\mathbf{v}} :=$ ExtractFilter$_\pi(\mathbf{v}, U \cup \{\mathbf{u}\}, G)$;
    foreach $p \in \pi$ do
        if $\alpha = \{a_{p \to \_}\}$ then
            $F[p] := F[p] \times \bar{a}_{p \to \_}.F_{\mathbf{v}}[p]$;
        else if $\alpha = \{a_{\_ \to p}\}$ then
            $F[p] := F[p] \times a_{\_ \to p}.F_{\mathbf{v}}[p]$;
        else
            $F[p] := F[p] \times F_{\mathbf{v}}[p]$;
if $rec[\mathbf{u}] =$ true then
    foreach $p \in \pi : X_{\mathbf{u}} \in fv(F[p])$ do
        $F[p] :=$ rec$(\mathbf{X_u})\, F[p]$;
return $F$;
```

7 Conclusion

Some research efforts on model checking web services have already been proposed [1,11,21,23]. The most related paper that we are aware of is by Nakajima [20] who introduces a lattice-based security labelling into BPEL in order to detect potential insecure information leakage. The paper discusses how both the safety and security aspects can be analyzed in a single framework using the model-checking verification techniques. The main difference with our approach is that the notion of security considered in [20] is built upon a simple lattice-based model for security labels. Instead, our approach deals with more flexible security policies which can be dynamically specified by the service participants. As far as correctness is concerned, [20] considers safety properties such as deadlock freedom and specific progress properties. Our model instead deals also with the property of livelock freedom.

In conclusion, we have developed a formal method for the analysis of both information flow security and compliance of contract service compositions. This is based on the characterization of such properties in terms of modal μ-calculus formulae. This allows us to use a model checker, like the NCSU Concurrency Workbench, in order to simultaneously check non-interference and compliance. An algorithm for adaptable service compositions is also proposed. It computes the greatest relevant filter fixing them.

References

1. Abouzaid, F., Mullins, J.: Model-checking Web Services Orchestrations using BP-calculus. Electronic Notes in Theoretical Computer Science 255, 3–21 (2009)
2. Basciutti, T.: Model-Checking Web Services. Master's thesis, Department of Computer Science, University Ca' Foscari of Venice (2010)
3. Bernardi, G., Bugliesi, M., Macedonio, D., Rossi, S.: A Theory of Adaptable Contract-Based Service Composition. In: Proc. of International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, Workshop on Global Computing Models and Technologies (GlobalComp 2008), pp. 327–334. IEEE Computer Society, Los Alamitos (2008)
4. Bravetti, M., Zavattaro, G.: Contract Compliance and Choreography Conformance in the Presence of Message Queues. In: Bruni, R., Wolf, K. (eds.) WS-FM 2008. LNCS, vol. 5387, pp. 37–54. Springer, Heidelberg (2009)
5. Bravetti, M., Zavattaro, G.: A Foundational Theory of Contracts for Multi-party Service Composition. Fundamenta Informaticae 89(4), 451–478 (2009)
6. Carpineti, S., Castagna, G., Laneve, C., Padovani, L.: A Formal Account of Contracts for Web Services. In: Bravetti, M., Núñez, M., Tennenholtz, M. (eds.) WS-FM 2006. LNCS, vol. 4184, pp. 148–162. Springer, Heidelberg (2006)
7. Castagna, G., Gesbert, N., Padovani, L.: A Theory of Contracts for Web Services. In: Proc. of the annual Symposium on Principles of Programming Languages (POPL 2008), pp. 261–272. ACM press, New York (2008)
8. Castagna, G., Gesbert, N., Padovani, L.: A Theory of Contracts for Web Services. ACM Transactions on Programming Languages and Systems (TOPLAS) 31, 53–61 (2009)
9. Clarke, E.M., Grumberg, O., Peled, D.A.: Model checking. The MIT Press, Cambridge (1999)
10. Cleaveland, R., Sims, S.: The NCSU Concurrency Workbench. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 394–397. Springer, Heidelberg (1996)
11. Dai, G., Bai, X., Zhao, C.: A Framework for Model Checking Web Service Compositions Based on BPEL4WS. In: Proc. of the IEEE International Conference on e-Business Engineering (ICEBE 2007), pp. 165–172. IEEE Computer Society, Los Alamitos (2007)
12. Focardi, R., Gorrieri, R.: A Classification of Security Properties for Process Algebras. Journal of Computer Security 3(1), 5–33 (1994/1995)
13. Focardi, R., Rossi, S.: Information Flow Security in Dynamic Contexts. Journal of Computer Security 14(1), 65–110 (2006)
14. Goguen, J.A., Meseguer, J.: Security Policies and Security Models. In: Proc. of the IEEE Symposium on Security and Privacy (SSP 1982), pp. 11–20. IEEE Computer Society, Los Alamitos (1982)
15. Kozen, D.: Results on the Propositional μ-calculus. Theoretical Computer Science 27, 333–354 (1983)
16. Laneve, C., Padovani, L.: The *must* Preorder Revisited. In: Caires, L., Vasconcelos, V.T. (eds.) CONCUR 2007. LNCS, vol. 4703, pp. 212–225. Springer, Heidelberg (2007)
17. Mader, A.: Modal μ-calculus, Model Checking, and Gauss Elimination. In: Brinksma, E., Steffen, B., Cleaveland, W.R., Larsen, K.G., Margaria, T. (eds.) TACAS 1995. LNCS, vol. 1019, pp. 72–88. Springer, Heidelberg (1995)
18. Milner, R.: Communication and Concurrency. Prentice Hall International Series in Computer Science, vol. 92. Prentice Hall, Englewood Cliffs (1989)
19. Müller-Olm, M.: Derivation of Characteristic Formulae. Electronic Notes in Theoretical Computer Science 18 (1998)

20. Nakajima, S.: Model-Checking of Safety and Security Aspects in Web Service Flows. In: Koch, N., Fraternali, P., Wirsing, M. (eds.) ICWE 2004. LNCS, vol. 3140, pp. 488–501. Springer, Heidelberg (2004)
21. Nakajima, S.: Model-Checking Behavioral Specification of BPEL Applications. Electronic Notes in Theoretical Computer Science 151, 89–105 (2006)
22. Ryan, P., Schneider, S.: Process Algebra and Non-Interference. Journal of Computer Security 9(1/2), 75–103 (2001)
23. Schlingloff, H., Martens, A., Schmidt, K.: Modeling and Model Checking Web Services. Electronic Notes in Theoretical Computer Science 126, 3–26 (2005)

Distributed Adaption of Dining Philosophers

S. Andova[1], L.P.J. Groenewegen[2,*], and E.P. de Vink[1]

[1] Department of Mathematics and Computer Science
Technische Universiteit Eindhoven, The Netherlands
[2] FaST-Group, LIACS, Leiden University, The Netherlands
luuk@liacs.nl

Abstract. Adaptation of a component-based system can be achieved in the coordination modelling language Paradigm through the special component McPal. McPal regulates the propagation of new behaviour and guides the changes in the components and in their coordination. Here we show how McPal may delegate part of its control to local adaptation managers, created on-the-fly, allowing for distribution of the adaptation indeed. We illustrate the approach for the well-known example of the dining philosophers problem, by modelling the migration from a deadlock-prone solution to a deadlock-free starvation-free solution without any system quiescence. The adaptation goes through various stages, exhibiting shifting control among McPal and its helpers, and changing degrees of orchestrated and choreographic collaboration.

1 Introduction

Many systems today are affected by changes in their operational environment when running, while they cannot be shutdown to be updated and restarted again. Instead, *dynamic adaptive systems* must be able to change their behaviour on-the-fly and to self-manage adaptation steps accommodating a new policy.

Dynamic adaptive systems consist of interacting components, usually distributed, and possibly hierarchically organized. In such a system, components may start adaptation in response to various triggers, such as changes in the underlying execution environment (e.g. failures or network congestion) or changes of requirements (e.g. imposed by the user). Adaptation of one component in the system may inadvertently influence the behavior of the components it is interacting with, possibly bringing about a cascade of dynamic changes in other parts of the system. Therefore, the adaptation of the system is a combination of local changes per component and global adaptation across components and hosts in the distributed system. As such, adaptation has to be performed in a consistent and coordinated manner so that the functionality of each separate component and of the system as a whole are preserved while the adaptation is in progress. Due to the complexity of the distributed dynamics of a system adapting on-the-fly, it may be rather difficult to understand whether a realization of a change

[*] Corresponding author.

plan indeed allows the system to perform as it is supposed to, and does not violate any of its requirements, during and after system adaptation.

One way to circumvent this is to formally model and analyze the system behaviour and the adaptation changes to be followed. In [13,3,6] we advocated how orchestrated adaptation can conveniently be captured in the coordination modeling language Paradigm. In Paradigm, a system architecture is organized along specific collaboration dimensions, called partitions. A partition is a well-chosen set of sub-behaviours of the local behaviour of a component, specifying the phases the component goes through when a protocol is executed. In the protocol, at a higher layer in the architecture, the component participates via its role, an abstract representation of the phases. A protocol manager coordinates the phase transfers for the components involved. In fact, in Paradigm, dynamic adaptation is modeled as just another collaboration protocol, coordinated by a special component *McPal* [13,3,6]. As progress within a phase is completely local to the component, the use of phase transfer instead of state transfer, is the key concept of Paradigm. This makes it possible to model, at the same time and separated from one another, both behavioural local changes per component, and global changes across architectural layers. The formal semantics of Paradigm then allows for a rigorous analysis of the adaptation policy [5], at the local level of the components as well as in the coordination of the changes across adaptive parts of the distributed system.

The suitability of Paradigm to model *distributed* adaptation strategies, extending our earlier centralized studies, is shown in this paper on a dining philosophers example. A deadlock-prone solution of the dining philosophers problem is taken as a source system, to be migrated to a target solution, both deadlock-free and starvation-free. Both systems are modeled in Paradigm, as is the migrating from source to target. As typical for dynamic adaptation in Paradigm, *McPal* regulates the propagation of new behaviour and guides the structural changes in the components and in their coordination. But here, although adding to the complexity of the solution, *McPal* delegates part of its control to local adaptation managers *McPhil*, one for each philosopher, while *McPal* keeps controlling them globally. Thus, we argue, the component-based character of the Paradigm language allows for modeling distributed adaptation: separate modeling of local strategies, coordinated by *McPal* as system adaptation manager. The main contribution of the paper is, it reveals the distributed potential of system adaptation within Paradigm. In Section 6 we elaborate on it.

Related work. In recent years a number of approaches has been proposed addressing several issues of dynamic system adaptation. Some of them [14,8,18] focus on adaptive software architectures, where functionalities, considered as black boxes, are connected via ports. Formal modeling of dynamic adaptation has been addressed in e.g. [16,2,10,22]. However, none of these approaches deal with distribution explicitly. In [11,12] dynamic adaptation is formally modeled by means of graph transformation. Although graph transformation techniques are well suited for distributed systems, there is no explicit focus on modeling distributed control for adaptation in the papers mentioned.

Some aspects of dynamic adaptation of distributed systems, tailored for the domains considered, have been treated in [1,17]. In the domain of Web Services, [1] proposes a method to generate distributed adapters from given service descriptions. [17] focuses on modeling and deployment of distributed resources for adaptive services in a mobile environment. A framework for formal modeling and verification of dynamic adaptation of distributed system, based on a transitional-invariant lattice technique, is proposed in [9]. The approach uses theorem proving techniques to show that during and after adaptation, the system always satisfies the transitional-invariants. This adaptation framework, however, does not support distribution in the style discussed in this paper: distributing adaptation tasks among local adaptation conductors by delegation.

The Conductor framework [21] for distributed adaptation allows for dynamic deployment of multiple adaptation conductors at various points in a network, an approach which is more suitable for complex and heterogeneous collaborations. It includes a distributed planning algorithm which determines for a triggered adaptation the most appropriate combination of conductors, distributed across the network. In [19] a distributed adaptation model for component-based applications is proposed. The model consists of two types of functionalities: mandatory that manage basic adaptation operations and optional that can be used to distribute adaptation activities. This way the adaptation mechanism of the whole system can be hierarchically organized, resembling as such our hierarchical structures of *McPal* conductors. However, both in [19] and [21], the main focus is on designing the adaptation itself, while the formal modeling and analysis of the adaptation remains uncovered, positioning them complementary to our treatment of distributed adaptation.

Structure of the paper. Section 2 is an overview of Paradigm through the example of the deadlock-prone solution as source system. The target system, deadlock and starvation free, is in Section 3. Section 4 gives the distributed migration set-up from source to target system, with Section 5 discussing coordination technicalities separately. Section 6 wraps up and provides conclusions.

2 Dining Philosophers As-Is: Deadlock-Prone

This section presents a first solution to the dining philosopher problem of five $Phil_i$ components sharing five $Fork_i$ components, $i = 1..5$. We shall refer to this solution as the *as-is* system or just *as-is*. The solution itself is the well-known and failing deadlock-prone solution: Any $Phil_i$, while thinking and getting hungry, first waits until the left $Fork_i$ can be got, then gets it, waits until the right $Fork_{i+1}$ can be got, gets it and once having both forks starts eating. After the eating has satisfied her hunger, $Phil_i$ lays down both forks and returns to thinking again. As an extra requirement, the as-is system has the ability to migrate from its ongoing as-is solution behaviour to to-be solution behaviour, unknown as yet but hopefully better than the failing as-is behaviour.

Apparently, steps taken by *Phil*s and step-like status changes of *Fork*s are to be consistently aligned in accordance to the particular as-is solution. This means, behaviour of the five *Phil*s and *Fork*s has to be coordinated such that the as-is solution is realized, failing as it may. Based on its capabilities for keeping ongoing behaviour constrained, the Paradigm language can specify coordination solutions not only for foreseen situations like the as-is system, but also for originally unforeseen migration to a still unknown to-be solution. Even more, Paradigm allows for really smooth migrations, i.e. with ongoing but gradually changing coordination during adaptation from as-is to to-be. In view thereof, one may have the special component *McPal* in a Paradigm model, at first not influencing the model at all, but hibernatingly present to guide upcoming system migration.

Through the example of the as-is solution for the five *Phil* and *Fork* components with a hibernating *McPal* in place, we shall briefly introduce Paradigm. The coordination modeling language Paradigm has five basic notions: STD, phase, (connecting) trap, role and consistency rule. For more elaborate introductions see [5] (in-depth) or [4] (more intuitive).

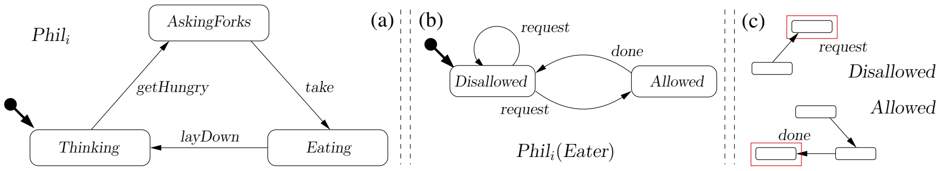


Fig. 1. The five $Phil_i$: (a) STD, (b) role *Eater*, (c) phase/trap constraints

Component behaviour is specified by *STD*s, state-transition diagrams. Figure 1a gives the STD for each $Phil_i$ in UML style. It says, $Phil_i$ starts in state *Thinking* and has forever cycling behaviour, passing through her three states by repeatedly taking her three actions *getHungry*, *take*, *layDown* in that order. When $Phil_i$ gets hungry in *Thinking*, she takes action *getHungry*, thus arriving in *AskingForks*. In accordance to the as-is solution, when an arbitrary $Phil_i$ is sojourning in state *AskingForks*, the following is supposed to happen subsequently: (*i*) $Fork_i$ is claimed by her; (*ii*) $Fork_i$ is assigned to her; (*iii*) $Fork_{i+1}$ is claimed by her; (*iv*) $Fork_{i+1}$ is assigned to her. Thereupon $Phil_i$ performs action *take* for taking up the two *Fork*s assigned to her by now from the table, thus arriving in state *Eating*. Later when no longer hungry, $Phil_i$ goes from *Eating* to *Thinking* by taking action *layDown* for returning both *Fork*s to the table. We see, claiming and assigning of *Fork*s is not reflected in the STD steps of $Phil_i$.

In Paradigm, such claiming and assigning is to be modeled through temporary constraints on STD behaviour; here on $Fork_i$ and on $Fork_{i+1}$ behaviour influenced by $Phil_i$, as we shall see below. What we can observe already, also $Phil_i$'s STD behaviour is like-wise influenced, i.e. temporarily constrained, by the combined behaviours of $Fork_i$ and $Fork_{i+1}$, as $Phil_i$ can proceed to state *Eating* only if both *Fork*s have been assigned to her and remain so. In addition, as long as the *Fork*s remain assigned to her, $Phil_i$ can return to *Thinking* but she should not be able to proceed to *AskingForks* while holding them.

In general, within Paradigm a component participating in a collaboration does not contribute to the collaboration via its STD behaviour directly, but via a so-called *role*. Such a role is a different, global STD for the component built on top of the original STD, dealing with the temporary constraints that are important to the collaboration. The role contributes relevant essence only, role-wise distilled from the more detailed local component behaviour.

Figure 1b specifies STD role $Phil_i(Eater)$ contributed by $Phil_i$ to the collaboration called $Phil2Forks_i$. States of role $Phil_i(Eater)$ are referred to as *phases* of the $Phil_i$ STD: temporarily valid behavioural constraints imposed on $Phil_i$. Figure 1b mentions two phases: *Disallowed* and *Allowed*. Figure 1c graphically couples the two phases to $Phil_i$, by representing each phase as a subSTD, a scaled-down part of $Phil_i$. As one can see, phase *Disallowed* (on top) prohibits $Phil_i$ to be in *Eating* but she may get as far as *AskingForks*. Contrarily, phase *Allowed* (at bottom) permits $Phil_i$ to enter and to leave *Eating* once, but returning to *AskingForks* is not allowed.

Phase drawings are additionally decorated with one or more polygons, each polygon grouping states of that phase. In the simple case of Figure 1 polygons are rectangles comprising a single state. Polygons visualize so-called *traps*: a trap, as a subset of states in a phase, once entered, cannot be left as long as the phase remains the constraint imposed. A trap serves as a guard for a phase transfer (in role STDs). Therefore, traps label transitions in a role STD, cf. Figure 1b: the guard marking the transition from the previous phase (it is a trap of) to a next phase. In such a case, where all states in a trap are indeed states of the next phase, the trap is called *connecting* from the previous phase to the next.

Thus, role $Phil_i(Eater)$ behaviour, see Figure 1b, expresses the ongoing alternation between *Disallowed* and *Allowed*: phase transfer from *Disallowed* to *Allowed* only happens after connecting trap *request* has been entered; similarly, phase transfer from *Allowed* to *Disallowed* only happens after connecting trap *done* has been entered. Moreover, an explicitly prolonged sojourn in *Disallowed* can happen after the (connecting) trap *request* has been entered.

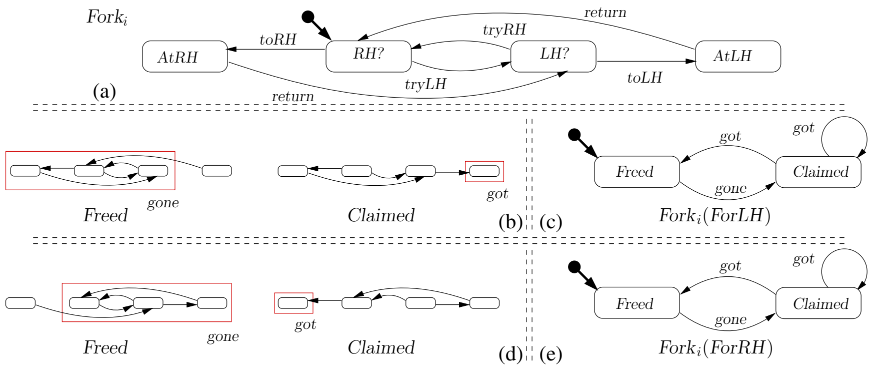


Fig. 2. Five $Fork_i$: (a) STD, (b,d) phase/trap constraints per (c,e) *ForLH*, *ForRH* role

The STD *Fork*$_i$ is visualized in Figure 2a. State *AtRH* means, *Fork*$_i$ is assigned to (the right hand of) *Phil*$_{i-1}$. Similarly, state *AtLH* means, *Fork*$_i$ is assigned to (the left hand of) *Phil*$_i$. To express not being assigned to any of the two philosophers *Phil*$_{i-1}$ or *Phil*$_i$, STD *Fork*$_i$ is on the table, but in two different states *LH?* and *RH?*, reflecting a different bias: in *LH?* the bias is to *Phil*$_i$ and in *RH?* the bias is to *Phil*$_{i-1}$. In addition, upon returning to the table from having been assigned to a philosopher's hand, the bias is first to the other philosopher. This means, each *Fork* follows a round robin approach for honouring requests from *Phil*$_{i-1}$ and *Phil*$_i$, rather than a nondeterministic one. Figure 2 also presents two roles of *Fork*$_i$: (c) role *ForLH* for collaborating with *Phil*$_i$ (left hand) and (e) role *ForRH* for collaborating with *Phil*$_{i-1}$ (right hand). Role *ForLH* is based on phases *Freed* and *Claimed* and their connecting traps *gone* and *got* as given in part 2b. Note for instance, how in *Freed* of role *ForLH* the particular *Fork*$_i$ is being steered towards giving up staying assigned to *Phil*$_i$'s left hand, thus returning to the table with the possibility to get assigned to *Phil*$_{i-1}$'s right hand but, for the moment, not to *Phil*$_i$'s left hand.

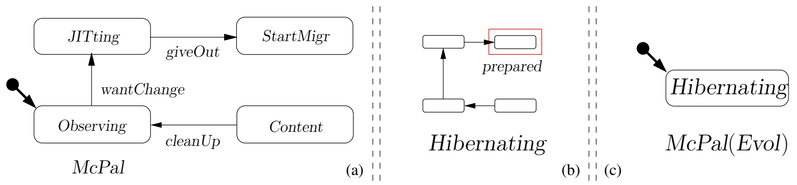


Fig. 3. *McPal*: (a) STD, (b) phase/trap constraint (b,d) role *McPal*(*Evol*)

The five *Phil*s and *Fork*s are all component ingredients needed for the as-is system. In view of still unknown later adaptation, an extra STD *McPal* is in place, see Figure 3a. Here *McPal* is in its hibernating form, not interfering at all with the as-is system, but with the ability to interfere with itself first, thus adapting itself and then later, as a consequence of its gained dynamics, to interfere with the as-is system. Figure 3bc underline this idea: (*i*) via phase *Hibernating* being the full *McPal* behaviour as long as *McPal* has not adapted itself yet; (*ii*) via role *Evol* which is restricted to sojourning in phase *Hibernating* as long as *McPal* remains unchanged. Thus we see, *McPal* starts in *Observing* and via *JITting* can go as far as *StartMigr*, which coincides with entering trap *prepared* of *Hibernating*. What cannot be seen from the figure but only from the consistency rules given below, through step *giveOut* leading into trap *prepared*, the hibernating *McPal* will extend the Paradigm as-is model specification with a specification of a to-be model as well as with a well-fitting model fragment for possible migration trajectories from as-is to to-be. To this aim, *McPal* embodies the reflectivity of a Paradigm model, by owning a local variable *Crs* where it stores the current model specification: consistency rules with all STDs, phases, traps and roles involved. Thus, by taking step *giveOut*, *Crs* will be extended, with at least one step series from *StartMigr* to *Content*, such that the no-longer-hibernating *McPal* is able to coordinate the various migration trajectories. Having returned to phase *Hibernating*, step *cleanUp* from *Content* to *Observing* then refreshes *Crs* by removing all model fragments obsolete by then, keeping the to-be model only. Note, so far *McPal* is the same as in [3,6].

In terms of the STDs, phases, traps and roles, Paradigm defines the 'coordination glue' between them through its notion of a *consistency rule*, being a synchronization of single role steps from different roles. Such a consistency rule may be coupled –additionally synchronized– with one detailed step of a so-called conductor component. Also local variables, such as *Crs*, can be updated. A consistency rule has as format: (*i*) it contains one asterisk $*$, with $*$'s right-hand side nonempty; (*ii*) optionally, at the left-hand side of $*$ it gives the one conductor step if relevant; (*iii*) at the right-hand side of $*$ it gives the listing of the role steps being synchronized; (*iv*) optionally, at the right-hand side $*$ a change clause can be given for updating variables. A consistency rule with a conductor step is called an *orchestration* step, a consistency rule without it is called a *choreography* step.

The set of consistency rules for the coordination of the as-is system, with *McPal* in place, is as follows.

$$* \; \mathit{Phil}_i(\mathit{Eater})\colon \mathit{Disallowed} \overset{\mathit{request}}{\rightarrow} \mathit{Disallowed},\; \mathit{Fork}_i(\mathit{ForLH})\colon \mathit{Freed} \overset{\mathit{gone}}{\rightarrow} \mathit{Claimed} \tag{1}$$

$$* \; \mathit{Fork}_i(\mathit{ForLH})\colon \mathit{Claimed} \overset{\mathit{got}}{\rightarrow} \mathit{Claimed},\; \mathit{Fork}_{i+1}(\mathit{ForRH})\colon \mathit{Freed} \overset{\mathit{gone}}{\rightarrow} \mathit{Claimed} \tag{2}$$

$$* \; \mathit{Fork}_{i+1}(\mathit{ForRH})\colon \mathit{Claimed} \overset{\mathit{got}}{\rightarrow} \mathit{Claimed},\; \mathit{Phil}_i(\mathit{Eater})\colon \mathit{Disallowed} \overset{\mathit{request}}{\rightarrow} \mathit{Allowed} \tag{3}$$

$$\begin{aligned} * \; & \mathit{Phil}_i(\mathit{Eater})\colon \mathit{Allowed} \overset{\mathit{done}}{\rightarrow} \mathit{Disallowed}, \\ & \mathit{Fork}_i(\mathit{ForLH})\colon \mathit{Claimed} \overset{\mathit{got}}{\rightarrow} \mathit{Freed},\; \mathit{Fork}_{i+1}(\mathit{ForRH})\colon \mathit{Claimed} \overset{\mathit{got}}{\rightarrow} \mathit{Freed} \end{aligned} \tag{4}$$

$$\mathit{McPal}\colon \mathit{JITting} \overset{\mathit{giveOut}}{\rightarrow} \mathit{StartMigr} \;*\; \mathit{McPal}\colon [\, \mathit{Crs} := \mathit{Crs} + \mathit{Crs}_{\mathit{migr}} + \mathit{Crs}_{\mathit{toBe}} \,] \tag{5}$$

$$\mathit{McPal}\colon \mathit{Content} \overset{\mathit{cleanUp}}{\rightarrow} \mathit{Observing} \;*\; \mathit{McPal}\colon [\, \mathit{Crs} := \mathit{Crs}_{\mathit{toBe}} \,] \tag{6}$$

It is through consistency rules (1)–(4) the deadlock-prone solution is achieved. Their choreographic specification can be read like this (numbers referring to rules): (1) if Phil_i wants to eat and her left Fork_i hasn't been claimed yet, it is claimed; (2) if Phil_i has got her left Fork_i assigned and her right Fork_{i+1} hasn't been claimed yet, it is claimed; (3) if Phil_i has got her right Fork_{i+1} assigned too, she is allowed to eat and can start doing so; (4) if Phil_i stops eating, her Fork_i and Fork_{i+1} are being freed and she is prohibited to eat any longer. In addition, rules (5)–(6) are orchestration steps with *McPal* as conductor, not influencing ongoing collaborative as-is behaviour, but extending the as-is model specification (5) and reducing the model specification to the to-be specification aimed at (6), after the migration has been done. The migration itself is not specified here, as neither the to-be situation nor intermediate migration are known at present. Please note, *Crs* is a variable of *McPal*. Similarly, both $\mathit{Crs}_{\mathit{migr}}$ and $\mathit{Crs}_{\mathit{toBe}}$ are variables containing consistency rules too, which means, their final value will be determined in view of the particular migration trajectory traversed.

3 Dining Philosophers To-Be: No Deadlock, No Starvation

Before addressing migration in Sections 4 and 5, this section presents the goal to be reached by the migration, referred to as the *to-be* system or *to-be* solution. The problem situation is the same as the one of the as-is situation, the five $Phil_i$ and five $Fork_i$. But, the solution is better now: no deadlock and also no starvation. This is achieved in the following well-known way: for at least one $Phil_i$, but not for all five, the order of claiming $Fork_i$ and $Fork_{i+1}$ is being reversed.

For the Paradigm model of the to-be solution this means, all STDs, phases, traps and roles remain the same, but the consistency rules are different. For their formulation we need some extra notation. Let the index sets L, R be a non-empty disjoint partitioning of $\{1..5\}$, L referring to those $Phil_i$s for which the left $Fork_i$ is claimed first, and R referring to those $Phil_i$s for which the right $Fork_{i+1}$ is claimed first. Here we use $i \in L$, $j \in R$.

$$* \; Phil_i(Eater)\colon Disallowed \overset{request}{\to} Disallowed,\; Fork_i(ForLH)\colon Freed \overset{gone}{\to} Claimed \tag{7}$$

$$* \; Fork_i(ForLH)\colon Claimed \overset{got}{\to} Claimed,\; Fork_{i+1}(ForRH)\colon Freed \overset{gone}{\to} Claimed \tag{8}$$

$$* \; Fork_{i+1}(ForRH)\colon Claimed \overset{got}{\to} Claimed,\; Phil_i(Eater)\colon Disallowed \overset{request}{\to} Allowed \tag{9}$$

$$* \; Phil_i(Eater)\colon Allowed \overset{done}{\to} Disallowed, \quad Fork_i(ForLH)\colon Claimed \overset{got}{\to} Freed,\; Fork_{i+1}(ForRH)\colon Claimed \overset{got}{\to} Freed \tag{10}$$

$$* \; Phil_j(Eater)\colon Disallowed \overset{request}{\to} Disallowed,\; Fork_{j+1}(ForRH)\colon Freed \overset{gone}{\to} Claimed \tag{11}$$

$$* \; Fork_{j+1}(ForRH)\colon Claimed \overset{got}{\to} Claimed,\; Fork_j(ForLH)\colon Freed \overset{gone}{\to} Claimed \tag{12}$$

$$* \; Fork_j(ForLH)\colon Claimed \overset{got}{\to} Claimed,\; Phil_j(Eater)\colon Disallowed \overset{request}{\to} Allowed \tag{13}$$

$$* \; Phil_j(Eater)\colon Allowed \overset{done}{\to} Disallowed, \quad Fork_j(ForLH)\colon Claimed \overset{got}{\to} Freed,\; Fork_{j+1}(ForRH)\colon Claimed \overset{got}{\to} Freed \tag{14}$$

$$McPal\colon\; JITting \overset{giveOut}{\to} StartMigr \; * \; McPal\colon [\, Crs := Crs + Crs_{migr} + Crs_{toBe} \,] \tag{15}$$

$$McPal\colon\; Content \overset{cleanUp}{\to} Observing \; * \; McPal\colon [\, Crs := Crs_{toBe} \,] \tag{16}$$

Rules (7)–(10) together with (15)–(16) are exactly the rules (1)–(6) from Section 2. It is not difficult to observe, rules (11)–(14) mirror (7)–(10) by reversing the order of claiming indeed. Furthermore, note that only the consistency rules have been adapted, so the change remains restricted to the ‘coordination glue’ between the components, particularly the choreography steps only. *McPal* is in hibernation, as usual with no migration going.

4 Migration Coordination Set-Up among Helpers

As Section 3 announced, the migration to be realized is from the as-is situation to the to-be situation, i.e. starting from the deadlock-prone solution of the dining philosophers problem to the well-known, far better deadlock-free and starvation-free solution, where at least one Phil_i gets her Fork_i and Fork_{i+1} assigned in a different order. So, there is ample room for different to-be solutions meeting the requirements. Also, for each to-be solution different migration trajectories towards it can be developed.

In view of this observation, we restrict the range of our to-be solutions as follows: regarding the sets L, R introduced above –claiming left fork first for Phil_i with $i \in L$ versus claiming right fork first for Phil_j with $j \in R$– we require L and R to have either 2 or 3 elements. Moreover, if $L = \{i, i'\}$ then Phil_i and $\mathit{Phil}_{i'}$ are not adjacent, i.e. $i = i' + 2$ or $i = i' + 3$, and similarly, if $R = \{j, j'\}$ then Phil_j and $\mathit{Phil}_{j'}$ are not adjacent. Thus, for the to-be solution, of the two groups of *Phil*s one group consists of two *Phil*s and the other group consists of three *Phil*s. In addition, the *Phil*s from the group of two are not neighbours. This reduction in admitted to-be solutions will illustrate the interplay of central change management and local change management more clearly. Moreover, it helps us in substantially restricting the range of migration trajectories, still showing the dynamic flexibility of the migration[1].

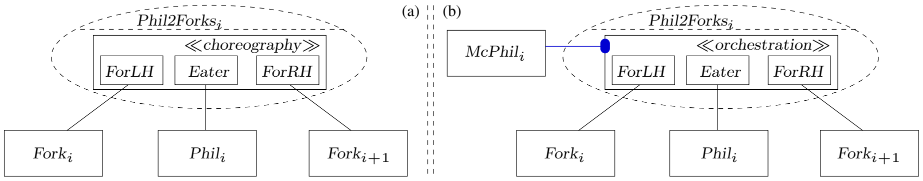


Fig. 4. Two collaboration snapshots (a) during hibernation, (b) during migration

Before addressing the actual migration through coordination not yet specified, we want to clarify an important structural difference in the collaboration of Phil_i and her two forks Fork_i and Fork_{i+1}: during *McPal*'s hibernation versus during migration. Figure 4a, in UML-style, gives collaboration diagram $\mathit{Phil2Forks}_i$ during hibernation. It says, the three components Phil_i, Fork_i, Fork_{i+1} are involved in it and, in line with the Paradigm model, they contribute to it via their respective roles *Eater*, *ForLH*, *ForRH*. This makes the collaboration into a choreography. Note, this architectural snapshot is valid for the as-is as well as for the to-be solution, the difference being in the behaviour only.

During the migration the collaboration has a slightly different structure, however, see Figure 4b. For each Phil_i an extra component McPhil_i is involved, meant as delegated helper of *McPal* for the $\mathit{Phil2Forks}_i$ collaboration only, to enlarge *McPal*'s influence. As we shall see below, McPhil_i joins the collaboration as a new local driver of the ongoing choreography, thereby turning $\mathit{Phil2Forks}_i$ into an orchestration with McPhil_i as its conductor, with essentially the same collaborative behaviour for a short while. Then, as conductor in place it migrates

[1] For an animated migration trajectory, see the extended version of the FACS 2010 presentation at `http://www.win.tue.nl/~evink/research/paradigm.html`.

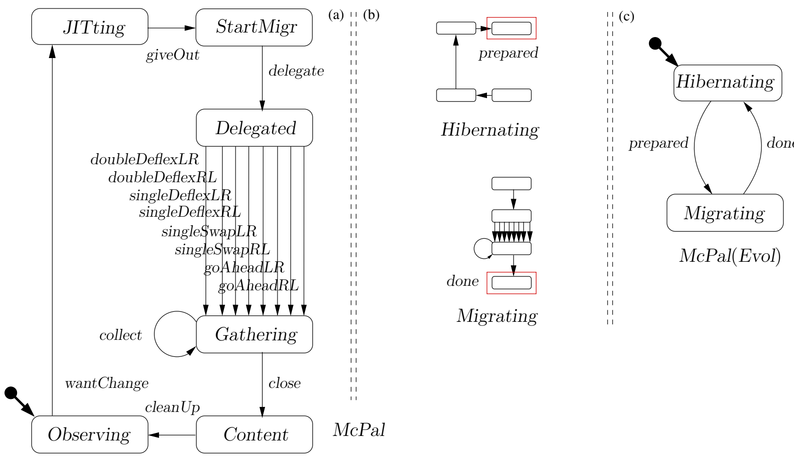


Fig. 5. *McPal* during migration: (a) STD, (b) phase/trap constraints, (c) role *Evol*

the orchestration and informs *McPal* about the result achieved so far, whereupon *McPal* decides about keeping or altering a result. Then $McPhil_i$ does so and steps back as conductor, turning collaboration $Phil2Forks_i$ into a choreography again.

McPal's actual migration activity is outlined in Figure 5: *McPal*, upon awakening from phase *Hibernating*, becomes active within phase *Migrating* as main conductor of the migration orchestration. Here it immediately delegates the actual migration coordination to its five helpers $McPhil_i$, by taking step *delegate*. In doing so, *McPal* provides each $McPhil_i$ with the orchestration rules for the local migration, while keeping the as-is choreography rules. Arrived in state *Delegated*, *McPal* then waits for the local results from the $McPhil_i$, being preliminary only. The preliminary results can be of two kinds: either $Phil_i$ (still) belongs to L or $Phil_i$ (now) belongs to R. Depending on the combined results of the five $McPhil_i$, conductor *McPal* takes one out of eight possible steps to state *Gathering*: by possibly letting zero, one or two specific $McPhil_i$ adjust their preliminary result when finalizing and by letting the other $McPhil_i$ make their preliminary results permanent. In state *Gathering*, *McPal* starts collecting the five sets $Crs_{i,toBe}$ of consistency rules the various $McPhil_i$ have to deliver when finalizing: the to-be choreography local to $Phil2Forks_i$, constituting $McPhil_i$'s final result. To this aim *McPal* takes step *collect* five times, one per $McPhil_i$. After having collected the five sets $Crs_{i,toBe}$ and after the five helper $McPhil_i$ have stopped their activities, *McPal* takes step *close* to state *Content*, thus entering trap *done* marking the final stage of the migration phase.

The overall migration conducting of *McPal* sets the stage for the local migration exerted by $McPhil_i$ on the ongoing collaboration $Phil2Forks_i$. The behaviour of each $McPhil_i$ is drawn in Figure 6a. From starting state *NonExisting* to *Awake* it takes step *stir*, to get ready for whatever it has to do. From *Awake* it takes step *takeOver* to state *JoiningIn*, thereby removing the as-is choreography rules from the (local) model specification Crs_i, thus keeping the orchestration

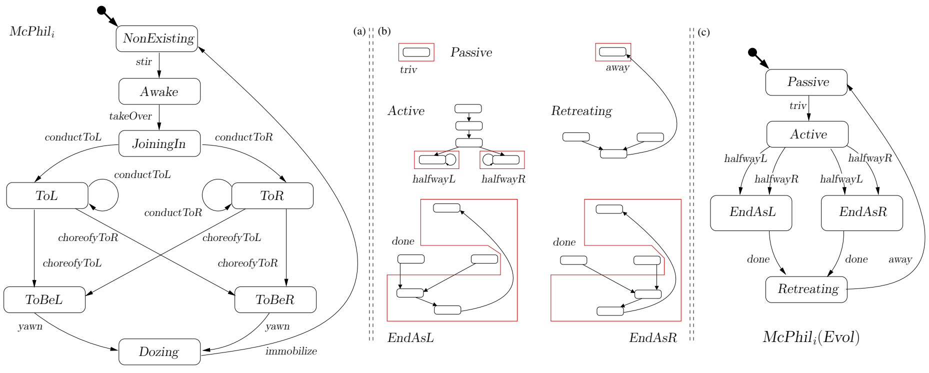


Fig. 6. *McPhil$_i$* during migration: (a) STD, (b) phase/trap constraints, (c) role *Evol*

rules only, that were already added earlier by *McPal* when delegating the local migration to *McPhil$_i$*.

By taking step *conductToL* from state *JoiningIn* to state *ToL* and by iterating step *conductToL* in *ToL*, helper *McPhil$_i$* sticks to the as-is orchestration, for the *L*-order that is. Similarly, by taking step *conductToR* from state *JoiningIn* to state *ToR* and by iterating step *conductToR* in *ToR*, helper *McPhil$_i$* swaps the orchestration of the as-is choreography to the orchestration for the *R*-order. From state *ToL* helper *McPhil$_i$* can, apart from iterating, either take step *choreofyToL* to state *ToBeL*, in which case *McPhil$_i$* sticks to the *L*-order but turns the orchestration back into the equivalent choreography, or *McPhil$_i$* can take step *choreofyToR* to state *ToBeR*, in which case *McPhil$_i$* swaps to the *R*-order (on second thought, instigated by *McPal*) and moreover turns the orchestration into the equivalent choreography for the *R*-order. Analogously, from state *ToR* helper *McPhil$_i$* can, apart from iterating, either take step *choreofyToR* to state *ToBeR*, in which case *McPhil$_i$* sticks to the *R*-order but turns the orchestration into the equivalent choreography, or *McPhil$_i$* can take step *choreofyToL* to state *ToBeL*, in which case *McPhil$_i$* swaps back to the *L*-order and moreover turns the orchestration into the equivalent choreography for the *L*-order. From then on, in two consecutive steps, viz. *yawn* and *immobilize*, helper *McPhil$_i$* returns to state *NonExisting*.

Figure 6b presents the phase and trap constraints on *McPhil$_i$*. Based on these constraints, role *McPhil$_i$*(*Evol*) is given in part 6c. In phase *Passive* helper *McPhil$_i$* can't do anything. In phase *Active* it can go as far as providing to *McPal* its preliminary result, being of two possible kinds, one per trap *halfwayL* or *halfwayR*. Phases *EndAsL* and *EndAsR* correspond to the two final results possible, the original *L*-order or the new *R*-order, respectively, available once trap *done* has been entered. Finally, in phase *Retreating* helper *McPhil$_i$* enters trap *away*. After that it returns to *Passive* where it can't do anything.

It is stressed all this is to happen dynamically, on-the-fly, without any system halting. Consistency rules specifying this turn out to be quite technical. Therefore we discuss them separately in Section 5.

5 Migration Coordination Distributed among Helpers

The consistency rules below specify the coordination according to Section 4's migration set-up. The technicalities of the rules mainly arise where change clauses manipulate sets of rules and model fragments aiming to influence the migration. Computing in terms of rules timely adapts the coordination strategy, gracefully enforcing the system's change. The following sets of consistency rules occur.

$\mathit{Crs}_{i,\mathit{asIs}}$	::=	choreography of $\mathit{Phil2Forks}_i$, L-order only (as in Section 2)
$\mathit{Crs}_{\mathit{hibr}}$	::=	orchestration conducted by McPal during phase $\mathit{Hibernating}$
$\mathit{Crs}_{\mathit{noHb}}$	::=	orchestration conducted by McPal during phase $\mathit{Migrating}$
$\mathit{Crs}_{i,\mathit{orch}}$	::=	orchestration conducted by McPhil_i
$\mathit{Crs}_{i,\mathit{toBeL}}$	::=	choreography of $\mathit{Phil2Forks}_i$ in L-order
$\mathit{Crs}_{i,\mathit{toBeR}}$	::=	choreography of $\mathit{Phil2Forks}_i$ in R-order
$\mathit{Crs}_{\mathit{migr}}$	::=	$\mathit{Crs}_{\mathit{noHb}} + \mathit{Crs}_{1,\mathit{orch}} + \cdots + \mathit{Crs}_{5,\mathit{orch}}$

The above sets are fixed, the sets below vary during the migration.

Crs	::=	varying orchestration/choreography, not governed by McPhil_i
Crs_i	::=	varying McPhil_i-governed rule set for $\mathit{Phil2Forks}_i$
$\mathit{Crs}_{i,\mathit{toBe}}$	::=	either $\mathit{Crs}_{i,\mathit{toBeL}}$ or $\mathit{Crs}_{i,\mathit{toBeR}}$
$\mathit{Crs}_{\mathit{toBe}}$	::=	growing from $\mathit{Crs}_{\mathit{hibr}}$ to $\mathit{Crs}_{\mathit{hibr}} + \mathit{Crs}_{1,\mathit{toBe}} + \cdots + \mathit{Crs}_{5,\mathit{toBe}}$

The fixed sets of the consistency rules are specified first. Note, the variable sets of consistency rules are updated through detailed change clauses involving the fixed sets.[2] Consistency rules (17)–(20) making up $\mathit{Crs}_{i,\mathit{asIs}}$ are exactly rules (1)–(4) from Section 2.

$$* \ \mathit{Phil}_i(\mathit{Eater})\colon \mathit{Disallowed} \overset{\mathit{request}}{\rightarrow} \mathit{Disallowed},\ \mathit{Fork}_i(\mathit{ForLH})\colon \mathit{Freed} \overset{\mathit{gone}}{\rightarrow} \mathit{Claimed} \tag{17}$$

$$* \ \mathit{Fork}_i(\mathit{ForLH})\colon \mathit{Claimed} \overset{\mathit{got}}{\rightarrow} \mathit{Claimed},\ \mathit{Fork}_{i+1}(\mathit{ForRH})\colon \mathit{Freed} \overset{\mathit{gone}}{\rightarrow} \mathit{Claimed} \tag{18}$$

$$* \ \mathit{Fork}_{i+1}(\mathit{ForRH})\colon \mathit{Claimed} \overset{\mathit{got}}{\rightarrow} \mathit{Claimed},\ \mathit{Phil}_i(\mathit{Eater})\colon \mathit{Disallowed} \overset{\mathit{request}}{\rightarrow} \mathit{Allowed} \tag{19}$$

$$\begin{aligned} * \ & \mathit{Phil}_i(\mathit{Eater})\colon \mathit{Allowed} \overset{\mathit{done}}{\rightarrow} \mathit{Disallowed}, \\ & \quad \mathit{Fork}_i(\mathit{ForLH})\colon \mathit{Claimed} \overset{\mathit{got}}{\rightarrow} \mathit{Freed},\ \mathit{Fork}_{i+1}(\mathit{ForRH})\colon \mathit{Claimed} \overset{\mathit{got}}{\rightarrow} \mathit{Freed} \end{aligned} \tag{20}$$

Likewise, rules (21)–(22) making up $\mathit{Crs}_{\mathit{hibr}}$, are exactly rules (5)–(6) and also rules (15)–(16) from Section 2 and 3, respectively.

$$\mathit{McPal}\colon \mathit{JITting} \overset{\mathit{giveOut}}{\rightarrow} \mathit{StartMigr} \ * \ \mathit{McPal}\colon [\, \mathit{Crs} := \mathit{Crs} + \mathit{Crs}_{\mathit{migr}} + \mathit{Crs}_{\mathit{toBe}} \,] \tag{21}$$

$$\mathit{McPal}\colon \mathit{Content} \overset{\mathit{cleanUp}}{\rightarrow} \mathit{Observing} \ * \ \mathit{McPal}\colon [\, \mathit{Crs} := \mathit{Crs}_{\mathit{toBe}} \,] \tag{22}$$

Note the two assignments to Crs. In rule (21), on the verge of migration, Crs is extended with the rules in $\mathit{Crs}_{\mathit{migr}}$ as well as in $\mathit{Crs}_{\mathit{toBe}}$, the latter set at this

[2] One may call this behaviour computation, programming in terms of behavioural constraints.

moment containing Crs_{hibr} only, already present in *Crs*. In rule (22), right after the migration, *Crs* is replaced by Crs_{toBe}, by then containing all choreography rules computed by the five *McPhil* plus the two rules in Crs_{hibr} already present.

Next we present the consistency rule set Crs_{noHb}, rules (23) to (36), covering the interaction of *McPal* and its five helper $McPhil_i$.

$$* \ McPal(Evol)\colon Hibernating \overset{prepared}{\rightarrow} Migrating \tag{23}$$

$$\begin{aligned} &McPal\colon StartMigr \overset{create}{\rightarrow} Delegated \ * \ McPal\colon [\, Crs := Crs_{noHb} + Crs_{hibr} \,], \\ &\quad McPhil_1(Evol)\colon Passive \overset{triv}{\rightarrow} Active, \ldots, McPhil_5(Evol)\colon Passive \overset{triv}{\rightarrow} Active, \\ &\quad McPhil_1[\, Crs_1 := Crs_{1,asIs} + Crs_{1,orch} \,], \ \ldots, \ McPhil_5[\, Crs_5 := Crs_{5,asIs} + Crs_{5,orch} \,] \end{aligned} \tag{24}$$

The set Crs_{noHb} contains the rule (23) for *McPal*'s own phase transfer from *Hibernating* to initiate the migration. From then on one finds orchestration rules for various conducting steps *McPal* may take within phase *Migrating*. Rule (24) gets the five helper $McPhil_i$ going, providing each with the local as-is choreographic rules as well as with its own orchestration rules, while *McPal* keeps those from Crs_{hibr} and Crs_{noHb} as its own rules only.

The STD of *McPal* in Figure 5 provides eight transitions from state *Delegated* to state *Gathering*. *McPal* takes a transition from its state *Delegated* to the state *Gathering* once all five $McPhil_i$ have reached a 'halfway' trap, either trap *halfwayL* or trap *halfwayR*, in their phase *Active*. Therefore, the figure shows eight different actions, dependent on the various combinations. By coupling a local transition of *McPal* to a global step in the *Evol* role of the $McPhil_i$, the proper transition is taken by *McPal* and the right continuation for the $McPhil_i$ is prescribed. Below, in the set of consistency rules Crs_{noHb}, we only provide the rules for the actions *doubleDeflexLR* and *singleDeflexRL*, rules (25) and (26), leaving the details of the remaining six rules to the reader.

$$\begin{aligned} &McPal\colon Delegated \overset{doubleDeflexLR}{\rightarrow} Gathering \ * \\ &\quad McPhil_1(Evol)\colon Active \overset{halfwayR}{\rightarrow} EndAsL, \ McPhil_2(Evol)\colon Active \overset{halfwayR}{\rightarrow} EndAsR, \\ &\quad McPhil_3(Evol)\colon Active \overset{halfwayR}{\rightarrow} EndAsL, \ McPhil_4(Evol)\colon Active \overset{halfwayR}{\rightarrow} EndAsR, \\ &\quad McPhil_5(Evol)\colon Active \overset{halfwayR}{\rightarrow} EndAsR \end{aligned} \tag{25}$$

$$\begin{aligned} &McPal\colon Delegated \overset{singleDeflexRL}{\rightarrow} Gathering \ * \\ &\quad McPhil_i(Evol)\colon Active \overset{halfwayR}{\rightarrow} EndAsR, \ McPhil_{i+1}(Evol)\colon Active \overset{halfwayL}{\rightarrow} EndAsL, \\ &\quad McPhil_{i+2}(Evol)\colon Active \overset{halfwayL}{\rightarrow} EndAsR, \ McPhil_{i+3}(Evol)\colon Active \overset{halfwayL}{\rightarrow} EndAsL, \\ &\quad McPhil_{i+4}(Evol)\colon Active \overset{halfwayL}{\rightarrow} EndAsL \end{aligned} \tag{26}$$

six more rules similar to (25) and (26) (27)–(32)

Rules (25)–(32) deal with the eight scenarios for handling the combined preliminary results from the five helper $McPhil_i$. In particular, rule (25) for action *doubleDeflexLR* covers the case where all five $McPhil_i$ follow *R*-order, so two non-neighbouring ones of them have to be swapped (back) to *L*-order, here we choose the first and the third. Similarly, rule (26) covers the case where exactly one $McPhil_i$ follows R-order, so another non-neighbouring one has to be swapped to R-order (as yet), here we choose $McPhil_{i+2}$.

$$\mathit{McPal}\colon \mathit{Gathering} \overset{\mathit{collect}}{\rightarrow} \mathit{Gathering} \;*\; \mathit{McPhil}_i(\mathit{Evol})\colon \mathit{EndAsR} \overset{\mathit{done}}{\rightarrow} \mathit{Retreating}, \quad \mathit{McPal}[\, \mathit{Crs} := \mathit{Crs} + \mathit{Crs}_i, \mathit{Crs}_{\mathit{toBe}} := \mathit{Crs}_{\mathit{toBe}} + \mathit{Crs}_{i,\mathit{toBe}}\,] \tag{33}$$

$$\mathit{McPal}\colon \mathit{Gathering} \overset{\mathit{collect}}{\rightarrow} \mathit{Gathering} \;*\; \mathit{McPhil}_i(\mathit{Evol})\colon \mathit{EndAsL} \overset{\mathit{done}}{\rightarrow} \mathit{Retreating}, \quad \mathit{McPal}[\, \mathit{Crs} := \mathit{Crs} + \mathit{Crs}_i, \mathit{Crs}_{\mathit{toBe}} := \mathit{Crs}_{\mathit{toBe}} + \mathit{Crs}_{i,\mathit{toBe}}\,] \tag{34}$$

$$\mathit{McPal}\colon \mathit{Gathering} \overset{\mathit{close}}{\rightarrow} \mathit{Content} \;*\; \mathit{McPhil}_1(\mathit{Evol})\colon \mathit{Retreating} \overset{\mathit{away}}{\rightarrow} \mathit{Passive}, \;\ldots,\; \mathit{McPhil}_5(\mathit{Evol})\colon \mathit{Retreating} \overset{\mathit{away}}{\rightarrow} \mathit{Passive} \tag{35}$$

$$*\; \mathit{McPal}(\mathit{Evol})\colon \mathit{Migrating} \overset{\mathit{done}}{\rightarrow} \mathit{Hibernating} \tag{36}$$

Rules (33) and (34) incorporate the final local R-order or the final local L-order, respectively, as final choreography part into the two variable sets *Crs* and Crs_{toBe}. Rule (35) passivates the five helper $McPhil_i$. Rule (36) allows *McPal* to return into hibernation, mirroring rule (23).

The set $Crs_{i,orch}$ with the actual adaptation orchestration by $McPhil_i$, comprising the rules (37) to (68) below, follows the STD of Figure 6.

$$\mathit{McPhil}_i\colon \mathit{Awake} \overset{\mathit{takeOver}}{\rightarrow} \mathit{JoiningIn} \;*\; \mathit{McPhil}_i[\, \mathit{Crs}_i := \mathit{Crs}_i - \mathit{Crs}_{i,\mathit{asIs}}\,] \tag{37}$$

$$\mathit{McPhil}_i\colon \mathit{JoiningIn} \overset{\mathit{conductToL}}{\rightarrow} \mathit{ToL} \;*\; \mathit{Phil}_i(\mathit{Eater})\colon \mathit{Disallowed} \overset{\mathit{request}}{\rightarrow} \mathit{Disallowed},\; \mathit{Fork}_i(\mathit{ForLH})\colon \mathit{Freed} \overset{\mathit{gone}}{\rightarrow} \mathit{Claimed} \tag{38}$$

$$\mathit{McPhil}_i\colon \mathit{JoiningIn} \overset{\mathit{conductToL}}{\rightarrow} \mathit{ToL} \;*\; \mathit{Fork}_i(\mathit{ForLH})\colon \mathit{Claimed} \overset{\mathit{got}}{\rightarrow} \mathit{Claimed},\; \mathit{Fork}_{i+1}(\mathit{ForRH})\colon \mathit{Freed} \overset{\mathit{gone}}{\rightarrow} \mathit{Claimed} \tag{39}$$

$$\mathit{McPhil}_i\colon \mathit{JoiningIn} \overset{\mathit{conductToR}}{\rightarrow} \mathit{ToR} \;*\; \mathit{Phil}_i(\mathit{Eater})\colon \mathit{Disallowed} \overset{\mathit{request}}{\rightarrow} \mathit{Allowed},\; \mathit{Fork}_{i+1}(\mathit{ForRH})\colon \mathit{Claimed} \overset{\mathit{got}}{\rightarrow} \mathit{Claimed} \tag{40}$$

$$\mathit{McPhil}_i\colon \mathit{JoiningIn} \overset{\mathit{conductToR}}{\rightarrow} \mathit{ToR} \;*\; \mathit{Phil}_i(\mathit{Eater})\colon \mathit{Allowed} \overset{\mathit{done}}{\rightarrow} \mathit{Disallowed},\; \mathit{Fork}_i(\mathit{ForLH})\colon \mathit{Claimed} \overset{\mathit{got}}{\rightarrow} \mathit{Freed},\; \mathit{Fork}_{i+1}(\mathit{ForRH})\colon \mathit{Claimed} \overset{\mathit{got}}{\rightarrow} \mathit{Freed} \tag{41}$$

$$\mathit{McPhil}_i\colon \mathit{JoiningIn} \overset{\mathit{conductToR}}{\rightarrow} \mathit{ToR} \;*\; \mathit{Phil}_i(\mathit{Eater})\colon \mathit{Disallowed} \overset{\mathit{request}}{\rightarrow} \mathit{Disallowed},\; \mathit{Fork}_i(\mathit{ForLH})\colon \mathit{Claimed} \overset{\mathit{got}}{\rightarrow} \mathit{Freed},\; \mathit{Fork}_{i+1}(\mathit{ForRH})\colon \mathit{Claimed} \overset{\mathit{triv}}{\rightarrow} \mathit{Claimed} \tag{42}$$

Rule (37) removes the as-is choreography. Here, (38)–(41), with $McPhil_i$ in *JoiningIn*, cover the four previous choreography steps of the as-is protocol, cf. rules (17–20), but now orchestrated. Rules (38) and (39) lead the conductor to state *ToL* to continue conducting the original L-order; rules (40) and (41) lead the conductor to state *ToR* to continue according to the new R-order. In these two steps the swap from L-order to R-order is easy as it happens to coincide with stopping to eat or with getting hungry anew. Rule (42) is needed to escape deadlock, a subtlety not further elaborated here.

$$\begin{aligned}
&\mathit{McPhil}_i\colon \mathit{ToR} \overset{\mathit{conductToR}}{\rightarrow} \mathit{ToR}\ * && (43)\\
&\quad \mathit{Phil}_i(\mathit{Eater})\colon \mathit{Disallowed} \overset{\mathit{request}}{\rightarrow} \mathit{Disallowed},\ \mathit{Fork}_{i+1}(\mathit{ForRH})\colon \mathit{Freed} \overset{\mathit{gone}}{\rightarrow} \mathit{Claimed}\\
&\mathit{McPhil}_i\colon \mathit{ToR} \overset{\mathit{conductToR}}{\rightarrow} \mathit{ToR}\ * && (44)\\
&\quad \mathit{Fork}_{i+1}(\mathit{ForRH})\colon \mathit{Claimed} \overset{\mathit{got}}{\rightarrow} \mathit{Claimed},\ \mathit{Fork}_i(\mathit{ForLH})\colon \mathit{Freed} \overset{\mathit{gone}}{\rightarrow} \mathit{Claimed}\\
&\mathit{McPhil}_i\colon \mathit{ToR} \overset{\mathit{conductToR}}{\rightarrow} \mathit{ToR}\ * && (45)\\
&\quad \mathit{Fork}_i(\mathit{ForLH})\colon \mathit{Claimed} \overset{\mathit{got}}{\rightarrow} \mathit{Claimed},\ \mathit{Phil}_i(\mathit{Eater})\colon \mathit{Disallowed} \overset{\mathit{request}}{\rightarrow} \mathit{Allowed}\\
&\mathit{McPhil}_i\colon \mathit{ToR} \overset{\mathit{conductToR}}{\rightarrow} \mathit{ToR}\ *\ \mathit{Phil}_i(\mathit{Eater})\colon \mathit{Allowed} \overset{\mathit{done}}{\rightarrow} \mathit{Disallowed}, && (46)\\
&\quad \mathit{Fork}_i(\mathit{ForLH})\colon \mathit{Claimed} \overset{\mathit{got}}{\rightarrow} \mathit{Freed},\ \mathit{Fork}_{i+1}(\mathit{ForRH})\colon \mathit{Claimed} \overset{\mathit{got}}{\rightarrow} \mathit{Freed}\\
&\text{four similar rules for cycling in } \mathit{ToL} && (47)\text{–}(50)
\end{aligned}$$

Rules (43)–(46), with *McPhil*$_i$ in *ToR*, cover the new *R*-order, basically implementing the to-be rules (11)–(14), but conducted by *McPhil*$_i$ while sojourning in state *ToR*, waiting for *McPal*'s consent. The symmetric rules (47)–(50), with *McPhil*$_i$ staying in *ToL* are suppressed.

$$\begin{aligned}
&\mathit{McPhil}_i\colon \mathit{ToL} \overset{\mathit{choreofyToL}}{\rightarrow} \mathit{ToBeL}\ * && (51)\\
&\quad \mathit{Phil}_i(\mathit{Eater})\colon \mathit{Disallowed} \overset{\mathit{request}}{\rightarrow} \mathit{Disallowed},\ \mathit{Fork}_i(\mathit{ForLH})\colon \mathit{Freed} \overset{\mathit{gone}}{\rightarrow} \mathit{Claimed},\\
&\quad \mathit{McPhil}_i\ [\,\mathit{Crs}_i := \mathit{Crs}_i - \mathit{Crs}_{i,\mathit{orch}} + \mathit{Crs}_{i,\mathit{toBeL}},\ \mathit{Crs}_{i,\mathit{toBe}} := \mathit{Crs}_{i,\mathit{toBeL}}\,]\\
&\mathit{McPhil}_i\colon \mathit{ToL} \overset{\mathit{choreofyToL}}{\rightarrow} \mathit{ToBeL}\ * && (52)\\
&\quad \mathit{Fork}_i(\mathit{ForLH})\colon \mathit{Claimed} \overset{\mathit{got}}{\rightarrow} \mathit{Claimed},\ \mathit{Fork}_{i+1}(\mathit{ForRH})\colon \mathit{Freed} \overset{\mathit{gone}}{\rightarrow} \mathit{Claimed},\\
&\quad \mathit{McPhil}_i\ [\,\mathit{Crs}_i := \mathit{Crs}_i - \mathit{Crs}_{i,\mathit{orch}} + \mathit{Crs}_{i,\mathit{toBeL}},\ \mathit{Crs}_{i,\mathit{toBe}} := \mathit{Crs}_{i,\mathit{toBeL}}\,]\\
&\mathit{McPhil}_i\colon \mathit{ToL} \overset{\mathit{choreofyToL}}{\rightarrow} \mathit{ToBeL}\ * && (53)\\
&\quad \mathit{Fork}_{i+1}(\mathit{ForRH})\colon \mathit{Claimed} \overset{\mathit{got}}{\rightarrow} \mathit{Claimed},\ \mathit{Phil}_i(\mathit{Eater})\colon \mathit{Disallowed} \overset{\mathit{request}}{\rightarrow} \mathit{Allowed},\\
&\quad \mathit{McPhil}_i\ [\,\mathit{Crs}_i := \mathit{Crs}_i - \mathit{Crs}_{i,\mathit{orch}} + \mathit{Crs}_{i,\mathit{toBeL}},\ \mathit{Crs}_{i,\mathit{toBe}} := \mathit{Crs}_{i,\mathit{toBeL}}\,]\\
&\mathit{McPhil}_i\colon \mathit{ToL} \overset{\mathit{choreofyToL}}{\rightarrow} \mathit{ToBeL}\ *\ \mathit{Phil}_i(\mathit{Eater})\colon \mathit{Allowed} \overset{\mathit{done}}{\rightarrow} \mathit{Disallowed}, && (54)\\
&\quad \mathit{Fork}_i(\mathit{ForLH})\colon \mathit{Claimed} \overset{\mathit{got}}{\rightarrow} \mathit{Freed},\ \mathit{Fork}_{i+1}(\mathit{ForRH})\colon \mathit{Claimed} \overset{\mathit{got}}{\rightarrow} \mathit{Freed},\\
&\quad \mathit{McPhil}_i[\,\mathit{Crs}_i := \mathit{Crs}_i - \mathit{Crs}_{i,\mathit{orch}} + \mathit{Crs}_{i,\mathit{toBeL}},\ \mathit{Crs}_{i,\mathit{toBe}} := \mathit{Crs}_{i,\mathit{toBeL}}\,]
\end{aligned}$$

Rules (51)–(54), with *McPhil*$_i$ moving from *ToL* to *ToBeL*, cover the installment of *L*-order conducting as the to-be protocol. In addition, all orchestration in *Crs*$_i$ is replaced by the *L*-order choreography.

$$\begin{aligned}
&\mathit{McPhil}_i\colon \mathit{ToL} \overset{\mathit{choreofyToR}}{\rightarrow} \mathit{ToBeR}\ *\ \mathit{Phil}_i(\mathit{Eater})\colon \mathit{Disallowed} \overset{\mathit{request}}{\rightarrow} \mathit{Disallowed}, && (55)\\
&\quad \mathit{Fork}_i(\mathit{ForLH})\colon \mathit{Freed} \overset{\mathit{triv}}{\rightarrow} \mathit{Freed},\ \mathit{Fork}_{i+1}(\mathit{ForRH})\colon \mathit{Freed} \overset{\mathit{gone}}{\rightarrow} \mathit{Claimed},\\
&\quad \mathit{McPhil}_i[\,\mathit{Crs}_i := \mathit{Crs}_i - \mathit{Crs}_{i,\mathit{orch}} + \mathit{Crs}_{i,\mathit{toBeR}}, \mathit{Crs}_{i,\mathit{toBe}} := \mathit{Crs}_{i,\mathit{toBeR}}\,]\\
&\mathit{McPhil}_i\colon \mathit{ToL} \overset{\mathit{choreofyToR}}{\rightarrow} \mathit{ToBeR}\ * && (56)\\
&\quad \mathit{Fork}_i(\mathit{ForLH})\colon \mathit{Claimed} \overset{\mathit{triv}}{\rightarrow} \mathit{Freed},\ \mathit{Fork}_{i+1}(\mathit{ForRH})\colon \mathit{Freed} \overset{\mathit{gone}}{\rightarrow} \mathit{Claimed},\\
&\quad \mathit{McPhil}_i[\,\mathit{Crs}_i := \mathit{Crs}_i - \mathit{Crs}_{i,\mathit{orch}} + \mathit{Crs}_{i,\mathit{toBeR}},\ \mathit{Crs}_{i,\mathit{toBe}} := \mathit{Crs}_{i,\mathit{toBeR}}\,]\\
&\mathit{McPhil}_i\colon \mathit{ToL} \overset{\mathit{choreofyToR}}{\rightarrow} \mathit{ToBeR}\ *\quad \mathit{Phil}_i(\mathit{Eater})\colon \mathit{Disallowed} \overset{\mathit{triv}}{\rightarrow} \mathit{Disallowed}, && (57)\\
&\quad \mathit{Fork}_i(\mathit{ForLH})\colon \mathit{Claimed} \overset{\mathit{triv}}{\rightarrow} \mathit{Freed},\ \mathit{Fork}_{i+1}(\mathit{ForRH})\colon \mathit{Claimed} \overset{\mathit{triv}}{\rightarrow} \mathit{Claimed},\\
&\quad \mathit{McPhil}_i\colon [\mathit{Crs}_i := \mathit{Crs}_i - \mathit{Crs}_{i,\mathit{orch}} + \mathit{Crs}_{i,\mathit{toBeR}},\ \mathit{Crs}_{i,\mathit{toBe}} := \mathit{Crs}_{i,\mathit{toBeR}}\,]
\end{aligned}$$

$$\begin{aligned}
&\mathit{McPhil}_i\colon \mathit{ToL} \overset{\mathit{choreofyToR}}{\rightarrow} \mathit{ToBeR}\ * && (58)\\
&\quad \mathit{Phil}_i(\mathit{Eater})\colon \mathit{Disallowed} \overset{\mathit{request}}{\rightarrow} \mathit{Allowed},\ \mathit{Fork}_{i+1}(\mathit{ForRH})\colon \mathit{Claimed} \overset{\mathit{got}}{\rightarrow} \mathit{Claimed},\\
&\quad \mathit{McPhil}_i[\,\mathit{Crs}_i := \mathit{Crs}_i - \mathit{Crs}_{i,\mathit{orch}} + \mathit{Crs}_{i,\mathit{toBeR}},\ \mathit{Crs}_{i,\mathit{toBe}} := \mathit{Crs}_{i,\mathit{toBeR}}\,]\\
&\mathit{McPhil}_i\colon \mathit{ToL} \overset{\mathit{choreofyToR}}{\rightarrow} \mathit{ToBeR}\ *\ \mathit{Phil}_i(\mathit{Eater})\colon \mathit{Allowed} \overset{\mathit{done}}{\rightarrow} \mathit{Disallowed}, && (59)\\
&\quad \mathit{Fork}_i(\mathit{ForLH})\colon \mathit{Claimed} \overset{\mathit{got}}{\rightarrow} \mathit{Freed},\ \mathit{Fork}_{i+1}(\mathit{ForRH})\colon \mathit{Claimed} \overset{\mathit{got}}{\rightarrow} \mathit{Freed},\\
&\quad \mathit{McPhil}_i[\,\mathit{Crs}_i := \mathit{Crs}_i - \mathit{Crs}_{i,\mathit{orch}} + \mathit{Crs}_{i,\mathit{toBeR}},\ \mathit{Crs}_{i,\mathit{toBe}} := \mathit{Crs}_{i,\mathit{toBeR}}\,]\\
&\text{nine rules for leaving } \mathit{ToR}\text{, similar to rules (51)–(59)} && (60)\text{–}(68)
\end{aligned}$$

Rules (55)–(59), with *McPhil*$_i$ heading for *ToBeR*, cover the orchestrated swapping from L-order to R-order, thereby installing it as choreography. In particular, (55) covers claiming the first (right) *Fork* without having to undo an earlier claim of the left *Fork*. Contrarily, (56) covers claiming the first (right) *Fork* together with necessary undoing of an earlier claim of the left *Fork*. Notably, rule (57) covers continuing to claim the first (right) *Fork* together with necessary undoing of an earlier claim of the left *Fork*. Thus, (57) provides an escape from deadlock, similar to rule (42). Rule (58) and (59) cover starting and stopping to eat, for the last time as resulting from L-order. The symmetric rules (60)–(68) for leaving *ToR* are omitted.

Finally, the rule sets $\mathit{Crs}_{i,\mathit{toBeL}}$ and $\mathit{Crs}_{i,\mathit{toBeR}}$ contain the choreography rules for the to-be-situation in L-order and in R-order, rules (69)–(72) and (73)–(76) respectively.

$$\begin{aligned}
&*\ \mathit{Phil}_i(\mathit{Eater})\colon \mathit{Disallowed} \overset{\mathit{request}}{\rightarrow} \mathit{Disallowed},\ \mathit{Fork}_i(\mathit{ForLH})\colon \mathit{Freed} \overset{\mathit{gone}}{\rightarrow} \mathit{Claimed} && (69)\\
&*\ \mathit{Fork}_i(\mathit{ForLH})\colon \mathit{Claimed} \overset{\mathit{got}}{\rightarrow} \mathit{Claimed},\ \mathit{Fork}_{i+1}(\mathit{ForRH})\colon \mathit{Freed} \overset{\mathit{gone}}{\rightarrow} \mathit{Claimed} && (70)\\
&*\ \mathit{Fork}_{i+1}(\mathit{ForRH})\colon \mathit{Claimed} \overset{\mathit{got}}{\rightarrow} \mathit{Claimed},\ \mathit{Phil}_i(\mathit{Eater})\colon \mathit{Disallowed} \overset{\mathit{request}}{\rightarrow} \mathit{Allowed} && (71)\\
&*\ \mathit{Phil}_i(\mathit{Eater})\colon \mathit{Allowed} \overset{\mathit{done}}{\rightarrow} \mathit{Disallowed}, && (72)\\
&\quad \mathit{Fork}_i(\mathit{ForLH})\colon \mathit{Claimed} \overset{\mathit{got}}{\rightarrow} \mathit{Freed},\ \mathit{Fork}_{i+1}(\mathit{ForRH})\colon \mathit{Claimed} \overset{\mathit{got}}{\rightarrow} \mathit{Freed}\\
&*\ \mathit{Phil}_i(\mathit{Eater})\colon \mathit{Disallowed} \overset{\mathit{request}}{\rightarrow} \mathit{Disallowed},\ \mathit{Fork}_{i+1}(\mathit{ForRH})\colon \mathit{Freed} \overset{\mathit{gone}}{\rightarrow} \mathit{Claimed} && (73)\\
&*\ \mathit{Fork}_{i+1}(\mathit{ForRH})\colon \mathit{Claimed} \overset{\mathit{got}}{\rightarrow} \mathit{Claimed},\ \mathit{Fork}_i(\mathit{ForLH})\colon \mathit{Freed} \overset{\mathit{gone}}{\rightarrow} \mathit{Claimed} && (74)\\
&*\ \mathit{Fork}_i(\mathit{ForLH})\colon \mathit{Claimed} \overset{\mathit{got}}{\rightarrow} \mathit{Claimed},\ \mathit{Phil}_i(\mathit{Eater})\colon \mathit{Disallowed} \overset{\mathit{request}}{\rightarrow} \mathit{Allowed} && (75)\\
&*\ \mathit{Phil}_i(\mathit{Eater})\colon \mathit{Allowed} \overset{\mathit{done}}{\rightarrow} \mathit{Disallowed}, && (76)\\
&\quad \mathit{Fork}_i(\mathit{ForLH})\colon \mathit{Claimed} \overset{\mathit{got}}{\rightarrow} \mathit{Freed},\ \mathit{Fork}_{i+1}(\mathit{ForRH})\colon \mathit{Claimed} \overset{\mathit{got}}{\rightarrow} \mathit{Freed}
\end{aligned}$$

Note, rules (17)–(76) cover all migration trajectories. The rather large number of sixty rules is the consequence of our aim to distribute the migration, thus revealing the distributed potential of the Paradigm-*McPal* tandem for system adaptation by giving freedom to *McPhil*s as delegates. As final remark we note, neither the STDs of *Phil*s and *Fork*s nor their roles *Eater*, *ForLH*, and *ForRH* roles had to be changed: the migration is fully situated within the coordination of the five ongoing collaborations. Again, *Phil* and *Fork* components remain running while the system migrates, dynamically indeed.

6 Discussion and Concluding Remarks

In the setting of component-based system development, we have addressed dynamic system adaptation without any form of quiescence. By using the coordination modeling language Paradigm, in combination with the special component *McPal*, we particularly underlined the suitability of the approach for dynamic adaptation in a distributed manner. The distributed potential of the Paradigm-*McPal* tandem is our main result, actually revealed through delegation among helpers. Concrete form to the distributive aspect is given via the dining philosophers example: letting the system adapt itself from a rather bad solution (deadlock) to a substantially better one having neither deadlock nor starvation.

In the context of the example, the distributed character of the adaption produces another three new results as spin-off, all three showing a wider reach of the approach: (*i*) creation/deletion of STDs, (*ii*) adaptation with self-healing, (*iii*) behaviour computation. We elaborate on the three of them first.

In line with the coordination features offered by Paradigm, distribution of adaptation is achieved through delegation. Moreover, as adaptation is towards an originally unforeseen to-be solution, delegation thereof is brought into action by *McPal*. This results in concrete delegation to originally unforeseen components $McPhil_i$, one per collaboration $Phil2Forks_i$. As the *McPhil* components exist neither at the time the as-is solution is ongoing with *McPal* in hibernation nor at the time the to-be solution is ongoing with *McPal* in hibernation, in this case we model both STD creation and STD deletion in Paradigm, at the start and at the end of *McPal*'s non-hibernating phase *Migrating*, respectively. Modeling creation and deletion is achieved by simulating it via the phases of the various $McPhil_i(Evol)$ roles: creation of $McPhil_i$ when bringing it to life by leaving phase *Passive*; deletion of $McPhil_i$ when taking its life by returning to phase *Passive*. This way, STDs for components and for their roles can easily be created and deleted in a dynamically consistent manner, as all this comes down to suitable coordination.

As explained at the start of Section 4, coordinating adaptation, referred to as migration, is being modeled in state *JITting* such that different to-be situations can be reached, possibly through different migration trajectories. Accordingly, the migration model distributes the migration coordination among five helper $McPhil_i$, with the initial aim of locally achieving a reasonable result. Then *McPal*, by centrally collecting the partial results and comparing them in state *Delegated*, redistributes additional, specific alignment directives among the same five helper $McPhil_i$. After execution of the directives, final results are gathered and compiled into the particular to-be solution arising from the distributed migration coordination effort. The self-healing aspect, explicitly present in this example, lies in the activities occurring in state *Delegated* in view of selecting one out of eight outgoing action-transitions to state *Gathering*: rules (25)–(32) specify which particular alignment has to be done. The selection decision is the self-healing: it is solely based on trap information, certain combinations of five *halfwayL* vs. *halfwayR* traps having been entered. This means, it is solely based on intermediate migration results. Only in case of the two actions *goAheadLR* or *goAheadRL* the self-healing is empty; in the six other cases there is at least one adjustment from L-order to R-order or vice versa, and often two. Please note, such adjustments indeed arise on-the-fly of the still ongoing migration. Also interesting to note is, the self-healing directives are given at the level of *McPal*, the self-healing directives are performed at the (lower) delegation level of the five helper $McPhil_i$, very much in line with the architectural ideas in [15].

The above form of self-healing is finalized in *McPal*'s state *Gathering*. There the final to-be model is compiled into Crs_{toBe}, through composition of smaller model fragments composed to that aim by each helper $McPhil_i$. Fragments are about behaviour, so their composition certainly is behaviour computation, at the level of *McPal* as well as at the level of each $McPhil_i$. Thus, our behaviour computation is a distributed computation.

Another interesting feature of the example is, the seamless zipping of a conductor into a choreography, turning it into an 'equivalent' orchestration. Conversely, the seamless zipping of a conductor out of an orchestration, turns it into the 'equivalent' choreography. In this perspective, the temporary conductor $McPhil_i$ is reminiscent to the notion of a 'scaffold' in [20]. In our example, through the additional *Evol* role of a conductor $McPhil_i$, the scaffold has additional flexibility, changing phase-wise, while the model remains ongoing during alterations as usual.

As one might have observed, quite some redundancy appears in the above. (*i*) Paradigm has it in the role concept, repeating essence of component dynamics in view of *exogenous coordination* via consistency rules. (*ii*) Two roles per *Fork* introduce even more redundancy in view of architectural separation of five collaborative concerns. Behavioural redundancy is present too, organized in line with the five collaboarations $Phil2Forks_i$: (*iii*) After any helper $McPhil_i$ has communicated its *partial result*, it possibly has to undo the partial result. Or (*iv*) $McPhil_i$ possibly does essentially nothing, as partial result and local as-is as well as local to-be collaboration remain unchanged (L: left fork first, as always). This means, within the *environment* of the other four ongoing collaborations, a single $McPhil_i$'s behaviour computation *robustly* meanders towards its final result instead of going there straightforwardly.

During the final panel session at FACS 2010 the above four italicized characteristics –robust instead of correct, environment as first class citizen, exogenous coordination, partial results– have been positioned [7] as crucial for service-orientation in comparison to component technology. They reflect the additional flexibility service-orientation has to offer, when taking the next step from component technology. In Paradigm, these characteristics arise from redundancy designed on purpose: in language, in model structure and in model dynamics.

Recently, the Paradigm-*McPal* tandem is being deployed within Edafmis. The ITEA-project Edafmis aims at innovative integration of ICT-support from different advanced imaging systems into non-standard medical intervention practice, such that all flexibility needed during such interventions can be sustained smoothly and quickly, adequately and pleasantly. Particularly, the possibility for distributed migrations, as presented here, is of great value.

As presenting our model uses the full size of the paper, we are not able to address formal verification and further analysis of the migration here. We do have some results already. In future work we will report on it in more detail, in combination with other interesting migrations of dining philosophers.

References

1. Alia, M., et al.: Managing distributed adaptation of mobile applications. In: Indulska, J., Raymond, K. (eds.) DAIS 2007. LNCS, vol. 4531, pp. 104–118. Springer, Heidelberg (2007)
2. Allen, R., Douence, R., Garlan, D.: Specifying and analyzing dynamic software architectures. In: Astesiano, E. (ed.) FASE 1998. LNCS, vol. 1382, pp. 21–37. Springer, Heidelberg (1998)
3. Andova, S., Groenewegen, L.P.J., Stafleu, J., de Vink, E.P.: Formalizing adaptation on-the-fly. In: Salaün, G., Sirjani, M. (eds.) Proc. FOCLASA 2009. ENTCS, vol. 255, pp. 23–44 (2009)
4. Andova, S., Groenewegen, L.P.J., Verschuren, J.H.S., de Vink, E.P.: Architecting security with Paradigm. In: de Lemos, R., Fabre, J.-C., Gacek, C., Gadducci, F., ter Beek, M. (eds.) Architecting Dependable Systems VI. LNCS, vol. 5835, pp. 255–283. Springer, Heidelberg (2009)
5. Andova, S., Groenewegen, L.P.J., de Vink, E.P.: Dynamic consistency in process algebra: From Paradigm to ACP. Science of Computer Programming, 45 (2010), doi:10.1016/j.scico.2010.04.011
6. Andova, S., Groenewegen, L.P.J., de Vink, E.P.: Towards dynamic adaptation of probabilistic systems. In: Margaria, T., Steffen, B. (eds.) ISoLA 2010 Part II. LNCS, vol. 6416, pp. 143–159. Springer, Heidelberg (2010)
7. Arbab, F.: Personal communication (2010)
8. Bencomo, N., et al.: Dynamically adaptive systems are product lines too: Using model-driven techniques to capture dynamic variability of adaptive systems. In: Proc. DSPL 2008, Limerick, pp. 23–32 (2008)
9. Biyani, K.N., Kulkarni, S.S.: Assurance of dynamic adaptation in distributed systems. Journal of Parallel Distributed Computing 68, 1097–1112 (2008)
10. Bradbury, J.S., et al.: A survey of self-management in dynamic software architecture specifications. In: Garlan, D., Kramer, J., Wolf, A.L. (eds.) Proc. WOSS 2004, pp. 28–33. ACM, New York (2004)
11. Bucchiarone, A., et al.: Self-repairing systems modeling and verification using agg. In: Proc. WICSA/ECSA 2009, pp. 181–190. IEEE (2009)
12. Ehrig, H., et al.: Formal analysis and verification of self-healing systems. In: Rosenblum, D.S., Taentzer, G. (eds.) FASE 2010. LNCS, vol. 6013, pp. 139–153. Springer, Heidelberg (2010)
13. Groenewegen, L., de Vink, E.: Evolution-on-the-fly with Paradigm. In: Ciancarini, P., Wiklicky, H. (eds.) COORDINATION 2006. LNCS, vol. 4038, pp. 97–112. Springer, Heidelberg (2006)
14. Kramer, J., Magee, J.: The evolving philosophers problem: dynamic change management. IEEE Transactions on Software Engineering 16, 1293–1306 (1990)
15. Kramer, J., Magee, J.: Self-managed systems: an architectural challenge. In: Briand, L.C., Wolf, A.L. (eds.) Proc. FOSE 2007, pp. 259–268. IEEE, Los Alamitos (2007)

16. Magee, J., Kramer, J.: Dynamic structure in software architectures. SIGSOFT Software Engineering Notes 21, 3–14 (1996)
17. Melliti, T., Poizat, P., Mokhtar, S.B.: Distributed behavioural adaptation for the automatic composition of semantic services. In: Fiadeiro, J.L., Inverardi, P. (eds.) FASE 2008. LNCS, vol. 4961, pp. 146–162. Springer, Heidelberg (2008)
18. Morin, B., et al.: An aspect-oriented and model-driven approach for managing dynamic variability. In: Czarnecki, K., et al. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 782–796. Springer, Heidelberg (2008)
19. Segarra, M.-T., André, F.: A distributed dynamic adaptation model for component-based applications. In: Awan, I., et al. (eds.) Proc. AINA 2009, pp. 525–529. IEEE, Los Alamitos (2009)
20. Stam, A.W.: Interaction Protocols in PARADIGM. PhD thesis, LIACS, Leiden University (2009)
21. Yarvis, M., Reiher, P., Popek, G.J.: Conductor: A framework for distributed adaptation. In: Proc. HOTOS 1999, Rio Rico, pp. 44–51. IEEE, Los Alamitos (1999)
22. Zhang, J., Goldsby, H.J., Cheng, B.H.C.: Modular verification of dynamically adaptive systems. In: Sullivan, K.J., et al. (eds.) Proc. AOSD 2009, pp. 161–172. ACM, New York (2009)

Component Service Promotion: Contracts, Mechanisms and Safety

Pascal André, Gilles Ardourel, and Mohamed Messabihi

AeLoS Team - LINA CNRS UMR 6241 - University of Nantes
2, rue de la Houssinière F-44322 Nantes Cedex, France
{FirstName.LastName}@univ-nantes.fr

Abstract. Composition is a core concept of component and service-based models. In hierarchical component composition, promotion is used to make services available at a higher level of the hierarchy without breaking encapsulation. In this article we will study different kinds of promotion of services equipped with contracts, their usefulness, as well as their safety by considering appropriate proof obligations. We introduce several explicit assertion constructs in order to reduce the proof effort. We study the impact of encapsulation and rich state description on these promotions. We illustrate the approach (specification and verification) with the Kmelia component language.

1 Introduction

Composition is a core concept of component models. In hierarchical component composition, *promotion* is used to make an entity available at a higher level of the hierarchy. The exposed entity may vary from one model to another: it can be a port, an interface or a service. In UML2 related component models such as Palladio, KobrA, Java/A [11], the exposed entity is a *port* and the promotion is obtained by a delegation connector which usually does not modify the sub-component features. The approach is similar in BIP [14] or in ADLs like Darwin and Unicon which model architectures as composite components [16]. Sofa [9] and Fractal [8] are component models that expose *interfaces*. Promotion is achieved by special interface bindings (delegate/subsume). Again the promoted interfaces usually remain unchanged except that connectors can be redefined. Last, *service* promotion can be seen as a special kind of service composition [15,13]. In this article we deal with this last category.

The motivation of our approach is to build correct components and composites considering both bottom-up and top-down construction approaches. The two guiding principles are abstraction and encapsulation: components are black boxes in horizontal composition (assembly) and vertical composition (composite). Therefore we aim at avoiding, detecting or correcting errors when assembling components and promoting services at the composite level. In previous works [3,2] we set the basis of building correct components and assemblies. The verification of correctness is done by establishing different proof obligations.

In this article we tackle the issue of the correct vertical composition and especially the problem of correct promotion. We show that promotion can be more flexible than it appears while still being safe. We present a verification method and show that the proof effort can be reduced by using explicit predicate constructs and capitalising from already-proven properties. We describe a model and a process that support scalability through the abstraction task that makes an assembly be seen as a component: we want to avoid the flattening of composites for both the specification and the verification tasks. As in our previous works, we propose a methodology equipped with tools which is applied in the context of the Kmelia component model [3].

The article is structured as follows. Section 2 introduces a general component model and the promotion correctness obligation is sketched. In Section 3 we give a classification of promotions in which we describe their uses and we distinguish between safe and unsafe promotions. The Section 4 is dedicated to the correctness verification of the kinds of promotion previously described, introducing predicate operators and illustrating their role in the proofs with a specific component language (Kmelia). In Section 5, we study the impact of encapsulation on the promotions. Section 6 is dedicated to related works, and finally Section 7 concludes the article.

2 Hierarchical Composition and Service Promotion

This section introduces a component model equipped with an internal composition operator that allows promotion of services. We then present promotion correctness based on the respect of the Client-Supplier Contract.

2.1 Simple Model and Example

Consider a component model that separates component interface from component body and enables component composition. Each component defines a state space (using typed variables V and invariant properties Inv) and services. A composite is a component that encapsulates other components (called subcomponents), which can themselves be composites. The component interface exposes services which therefore can be promoted at the composite level. The *provided* services (server point of view) of a component are distinct from its *required* services (client point of view). A service refers to a software functionality, defined here by a signature and pre/post-conditions. These are predicates over the state space variables and parameters (a *virtual* state space for a required service). Figure 1 illustrates the formalism used in this article. The boxes denote components and the grey (resp. white) "funnel" denotes provided (resp. required) services. Three components are considered: a composite cc that includes two components ocA and ocB. ocA provides a service provServ1 that is required by service reqServ1 of component ocB: an assembly link binds both services. ocB provides a service origProvServ, which is promoted at the composite cc level by promoProvServ: a promotion link binds both services.

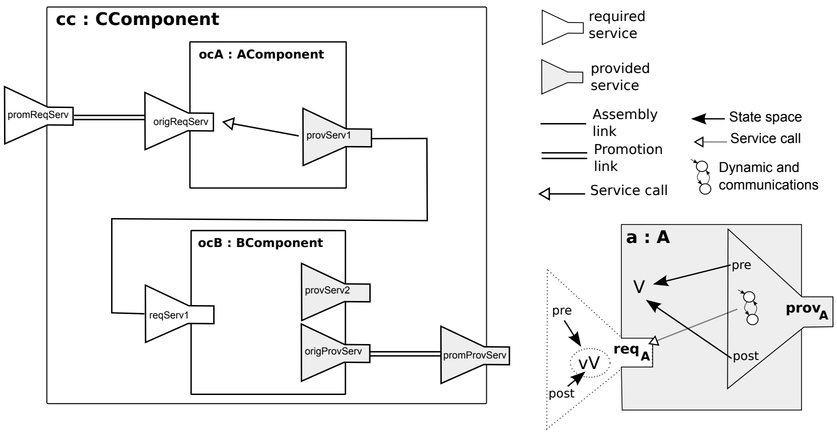


Fig. 1. Component and Composite component

Service promotion is usually associated with both the operations of making a service available at the composite level and adding it in the composite's interface. In this article we call **promotion** the former operation. The latter is only considered when explicitly mentioned.

2.2 Promotion Correctness

Let us focus on the promotion of a service *origin* provided by a component C to a service *promoted* of a composite CC. The contract of the service *origin* ensures that under the pre-condition Pre_{origin} of *origin* and the invariant inv_C of C, the service *origin* will satisfy its post-condition $Post_{origin}$ and preserve the invariant inv_C. Assuming the component state $before$ and $after$ values, we express this contract as follows:

Service Client-Supplier Contract

$Pre_{origin}(before, param) \wedge inv_C(before) \Rightarrow Post_{origin}(after, result) \wedge inv_C(after).$

In such a contract, the pre-condition should be established by the caller of the service (the client), while the post-condition and the invariant are established by the callee (the supplier). Since we handle only promotion in this article, the above contract holds for the service *origin*[1]. Also the actual service behaviour is assumed to be consistent with its contract, the possibility that the service has been underspecified is ignored and we do not try to make its pre/post-conditions more precise. Under these hypotheses, the promotion of *origin* to *promoted* is considered to be correct if the *service client-supplier contract* holds for *promoted*.

Whether choosing a top down or a bottom up approach, the designer selects from the set of available components which ones will be adequate for making its composite component. The services available in these components were not built to meet a specific composite designer's needs: even if they fit its purpose, they

[1] In [3] we showed how to prove such a contract in the Kmelia specification language.

can range from being quite generic operations with weak pre/post-conditions, to very specialised services with strong and precise pre/post-conditions. The composite designer has multiple and sometimes conflicting goals:

1. *safety* requires proving the composition of components to be safe,
2. *scalability* requires higher level abstractions and hiding the subcomponent structure and services,
3. *flexibility* requires taking into account future changes in the composite (for instance when a subcomponent is expected to be replaced by a better one),
4. environment *adaptability* requires tailoring the services to the environment the component is targeting, that might require uniform interfaces (which also enhance readability).

In order to achieve safety in the simplest way, most component models leave pre and post-conditions unchanged when promoting a service. However, it is necessary to change them to take the all the above goals into account. In the following, we will see which changes can be safely applied.

2.3 Changing Signatures and Predicates

The service signature can be seen as a part of a service's contract: the parameter types are part of the pre-condition and the return type is part of the post-condition. Since being of type T implies being of its super-type U, we can see that using a subtype in a predicate is strengthening it while using a super-type is weakening it. During promotion, the pre/post-conditions can be either

- weakened ($Predicate_{origin} \Rightarrow Predicate_{promoted}$),
- unchanged ($Predicate_{origin} \Leftrightarrow Predicate_{promoted}$),
- or strengthened ($Predicate_{promoted} \Rightarrow Predicate_{origin}$).

If none of these possibilities holds, then a part of the *promoted* predicate is not related to the *origin* predicates, and nothing can be concluded except that the predicate makes no sense and is considered to be incorrect.

3 Small Classification of Service Promotion

We study here different combinations: weakening, strengthening and unchanging, in order to capture their intent and to discuss their safety. In the sense of type systems, the original provided service can be seen as overriding the promoted one, which is type safe when it is contravariant on the parameters and covariant on the return type (and the opposite for required services). However we target a wider set of change possibilities.

3.1 Provided Service Promotion

Table 1 summarises the different changes of predicates during the promotion of a provided service and their safety.

- *Weakening pre-condition* is unsafe in the general case because it allows to break the Service Contract. A potential caller may invoke the *promoted* service in situations where the *origin* service can not ensure its post-condition.
- *Strengthening pre-condition* ensures that the original pre-condition will hold.
- *Weakening post-condition* is safe because the original service will then ensure more than it needs.
- *Strengthening post-condition* is unsafe in the general case. However, this part of the contract being the responsibility of the specifier of *promoted*, there are some cases in which it can put constraints on the execution context of *origin* in order to ensure it. We will characterise these cases in the following.

Table 1. Modifications of predicates when promoting a provided service

	Weakened Pre	Unchanged Pre	Strengthened Pre
Weakened Post	*Unsafe*	**Safe**	**Safe**
Unchanged Post	*Unsafe*	**Safe**	**Safe**
Strengthened Post	*Unsafe*	Generally Unsafe	Generally Unsafe

Out of the nine combinations derivable from these cases, we will not consider the cases involving the weakening of a pre-condition since they definitely cannot be proven consistent in the context of the composite alone.

Safe Kinds of Provided Service Promotion. The promotion kinds in this category are always safe provided that the contract at the composite level has been proven (this is discussed in Section 4.2).

1. *Keeping both the pre- and post-conditions* is the standard promotion that preserves the original contracts.
2. *Keeping the pre-condition and weakening the post-condition* is useful when the composite does not need as much precision in the type of a result as offered by the original service. Having a weaker post-condition will allow for easier future service substitutions. From a methodological point of view, it is related to information hiding, since it hides from the user, specific informations that are not deemed relevant in order to ease internal changes.
3. *Strengthening the pre-condition and keeping the post-condition* is useful when the composite *CC* is designed to evolve in an environment which is more constrained in terms of datatypes (e.g. sub-typing, domain restriction...) than the more generic components it contains. In such cases, strengthening the pre-condition to match those of the other services makes a more consistent interface and allows for substitutions with more strict services.
4. *Strengthening the pre-condition and weakening the post-condition* combines the previous goals and uses.

Generally Unsafe Promotion Kinds. Almost unsafe in the general case, some promotion kinds in this category are both safe and useful.

1. *Strengthening both the pre/post-conditions: "strengthening by parameters"* is safe if we can prove that the restriction on the parameters of the pre-condition implies the promoted post-condition. This case is illustrated in Section 4. While unlikely, the post-condition could also be strengthened by the context (see below).
2. *Keeping the pre-condition and strengthening the post-condition: "strengthening by context"* is safe only when the usage of the components in the composite strengthen the invariant on the state in such a way that it also strengthen a post condition depending on this state. This kind of promotion can only arise using a component model that supports state observability (see section 5). The context can only be restricted before the invocation of the promoted service, otherwise the component invariant can not be strengthened. This means that strengthening by context is restricted by a protocol (*e.g.* a user guide) or a specific initialisation of the sub-component.

We can illustrate the last case by considering a variant of the previous one where the constrained parameter is replaced by a state which is assigned a value either at the initialisation of the component by the composite (if it is supported by the component model) or by using a service which is called before the service *origin* (this can be ensured either by a protocol on the component C or by a protocol on the composite).

We saw earlier that a service's signature contains both a pre- and a post-condition. However, the post-condition strengthening will be more rare and harder to prove because type dependency is seldom expressed in post-conditions. To express it, one can use either anchor types or an explicit return of a parameter by reference modified by a service; the former requiring some kind of clone operation and the latter being rather unorthodox.

3.2 Required Service Promotion

In the specification of a required service, the pre-condition states what the potential callers inside the component will ensure upon calling the service which will achieve the required service. Conversely, the post-condition states what these callers expect from the required service. In this contract, the client is the caller of the required service (see the service call arrows in the right part of Fig. 1) and the supplier is the service that achieves it. This is the reverse situation of the one described in Table 1, with an additional restriction: the execution context in this case is in the behaviour of the callers and can not be constrained efficiently to enable generally unsafe situations.

1. *Keeping both the pre- and post-conditions* is the standard service delegation.
2. *Keeping the pre-condition and strengthening the post-condition* is requiring more than what was expected by the *origin* service. This can be done for uniformising the component interface.
3. *Weakening the pre-condition and keeping the post-condition* is promising less than the *origin* service *e.g.* to anticipate a change in *origin* service or to hide information.

4. *Weakening the pre-condition and strengthening the post-condition* accepts (2) and (3) together.

Table 2 summarises the different changes of assertions during the promotion of a required service and their safety.

Table 2. Modifications of assertions when promoting a required service

	Weakened Pre	Unchanged Pre	Strengthened Pre
Weakened Post	*Unsafe*	*Unsafe*	*Unsafe*
Unchanged Post	**Safe**	**Safe**	*Unsafe*
Strengthened Post	**Safe**	**Safe**	*Unsafe*

3.3 N-ary Service Promotion

In many component models, a set of $origin_i$ services can be promoted to a single *promoted* service. These kinds of promotion have different name or semantics: shared, multi-cast, gather-cast [4] but the consequences are simple: the client service contract is to be verified for each couple ($origin_i$, *promoted*).
For provided services:

– the pre-condition of *promoted* should imply the strongest of the $origin_i$ pre-conditions (if any) or their conjunction (if satisfiable),
– the post-condition of *promoted* should be implied by the weakest of the $origin_i$ post-conditions (if any) or their disjunction.

For required services:

– the pre-condition of *promoted* should be implied by the weakest of the $origin_i$ pre-conditions (if any) or their disjunction,
– the post-condition of *promoted* should imply the strongest of the $origin_i$ post-conditions (if any) or their conjunction (if satisfiable).

4 Verification Methodology of Promotion Correction

We illustrate the above promotion cases on a stock Management case study. The specification language is Kmelia [3]. Kmelia is an abstract formal component model dedicated to the specification and development of correct components. A Kmelia *component* is a container of services; it has a state space constrained by an invariant. A *service* is more than a simple operation; it has pre/post-conditions and a behaviour described with a labelled transition system (LTS). An *assembly* is a set of components linked via their required and provided services with the aim to build effective functionality. A *composite component* is a component that encapsulates an assembly. Kmelia is supported with an Eclipse-based analysis platform called COSTO. The proofs of correctness are experimented with AtelierB[2] one of the tool support of the B method [1]. The B method is a general purpose proof-based formal method; its input formalism is based on set-theory and first order logic.

[2] `http://www.aterlierb.eu`

4.1 An Example in Kmelia

The support example is a simplified *Stock Management* application. The system is designed by assembling two components: sm:StockManager and ve:Vendor. The former is the core business component to manage references and storage. The latter is the system access interface. The system specification is given in [3] and we go one step further in the specification by detailing the StockManager.

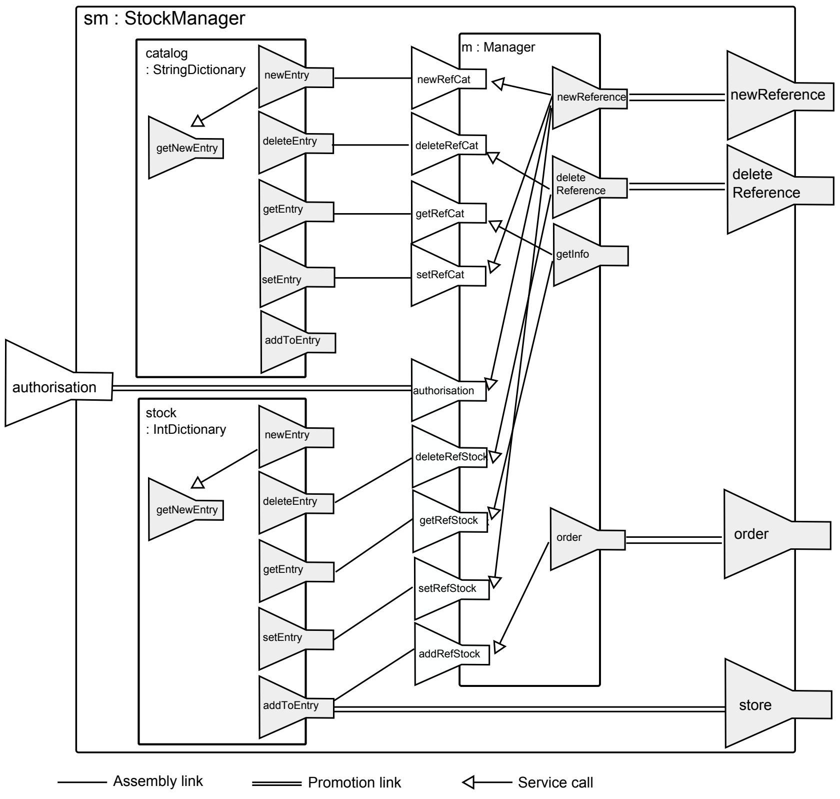


Fig. 2. Stock Manager composite component

Figure 2 shows that the composite component is made of three components: stock and catalog are instances of a specialised COTS[3] Dictionary, and m an instance of Manager is a controller of the composite. Most promoted services come from m. Listing 1 gives the specification of the StockManager composite. The first part describes the composite as a component while the **COMPOSITION** clause describes the encapsulated assembly (its components and assembly links). The From keyword denotes a promotion link for state variables or services. The

[3] component of-the-shelf

obs keyword characterises observable features, according to the rules given in Section 5. The promotion store−addToEntry is a generally unsafe situation of change: only natural numbers are possible.

Listing 1. StockManager composite

```
COMPONENT StockManager
INTERFACE
   provides : {newReference, deleteReference, store, order}
   requires : {authorisation}
USES {STOCKLIB}
VARIABLES
   obs catalog From m.keys;
   plabels From catalog.values; //product description
   pstock  From stock.values   //product quantity
INVARIANT
   obs @borned: size(catalog) <= maxRef,
   @referenced: forall ref : Reference | includes(catalog,ref) implies
      (plabels[ref] <> emptyString and plabels[ref] <> noQuantity),
   @notreferenced: forall ref : Reference | excludes(catalog,ref) implies
      (plabels[ref] = emptyString and plabels[ref] = noQuantity)
SERVICES
########### promoted provided services
provided newReference From m.newReference
End
provided store(pid : Integer; pqty : Integer) : Integer From stock.addToEntry
   RedefinedPre Stronger With (0 < pqty)
   RedefinedPost Stronger With pstock[pid] > old(pstock)[pid]
End
...
########### required services (partial description)
required authorisation() From m.authorisation
End
END_SERVICES
COMPOSITION
   Assembly
      Components
         catalog : StringDictionary; stock : IntDictionary; m : Manager;
      Links ////////////assembly links//////////
         @nrc: m.newRefCat catalog.newEntry
         @grc: m.getRefCat catalog.getEntry
         @grs: m.getRefStock stock.getEntry
         @srs: m.setRefStock stock.setEntry
         @ars: m.addRefStock stock.addToEntry
         ...
   End // assembly
END_COMPOSITION
```

During the edition of the specification with the COSTO platform, the first steps of analysis are performed on the specification: syntax, type-checking, structure and visibility well-formedness. The proofs on contracts are performed using B tools [3,2]. An *abstract machine* made of a state space (described by an invariant) and operations, is the unit of B specifications. The B method generates proof obligations to establish the consistency of abstract machines. Refinement of abstract machines to executable codes may also be considered. In the following we explain the properties to be checked which are related to promotion contracts, and the way we prove them using AtelierB. First, B machines are extracted for each kind of property we target, this extraction step is followed by processing proof obligations in AtelierB.

4.2 Explicit Promotion Operators

A promotion is *correct* if we can prove the *Client Service Contract* for the *promoted* service. For some kinds of promotion, we can capitalize from the proof of the service contract of *origin* and simplify greatly the problem of promotion correction. However, knowing which kind of promotion is being used is more difficult in the general case than the correction problem itself. In order to make the intent explicit, we propose to use predicate constructors to strengthen or to weaken an *origin* predicate. Here are the predicate constructors in the form of keywords WEAKER, STRONGER, WITH preceding a predicate.

- WEAKER $Predicate_{promoted}$,
  Introduces $Predicate_{promoted}$ as a weakened form of the original one. The associated proof obligation is: $(Predicate_{origin} \Rightarrow Predicate_{promoted})$.
- WEAKER WITH $PredicateW_{promoted}$,
  Constructs $Predicate_{promoted}$ as $Predicate_{origin} \vee PredicateW_{promoted}$ The associated proof obligation: $(Predicate_{origin} \Rightarrow Predicate_{promoted})$ holds by construction. The optional (w.r.t. safety) proof obligation is: $\neg\ (Predicate_{origin} \Rightarrow PredicateW_{promoted})$
- STRONGER $Predicate_{promoted}$,
  Introduces $Predicate_{promoted}$ as a strengthened form of the original one. The associated proof obligation is: $(Predicate_{promoted} \Rightarrow Predicate_{origin})$
- STRONGER WITH $PredicateW_{promoted}$,
  Constructs $Predicate_{promoted}$ as $Predicate_{origin} \wedge PredicateW_{promoted}$ The associated proof obligation: $(Predicate_{promoted} \Rightarrow Predicate_{origin})$ holds by construction. There is an optional (w.r.t. safety) proof obligation: satisfiability of $Predicate_{promoted}$

Table 3 summarises the impact of different changes of predicates during the promotion of a provided service and their safety. Using these constructors captures the designer intent, reduces the proof effort (when the contract is satisfied by construction or when it is always unsafe) and in some cases allows to automatically detect errors without resorting to a proof assistant (*e.g.* weakening pre-conditions in a provided service, strengthening post-conditions which do not depend on constrained state or parameters). An additional case of WEAKER WITH can also be used in models that support predicate names by removing a specific part of a conjunction predicate, which ensures the weakening by construction.

Table 3. Modifications of predicates when promoting a provided service

Pre	Weakened	Strengthened	Strengthened with	Unchanged
Unchanged Post	*Unsafe*	**Safe**	**Proven Safe**	**Proven Safe**
Weakened Post With	*Unsafe*	**Safe**	**Proven Safe**	**Proven Safe**
Weakened Post	*Unsafe*	**Safe**	**Safe**	**Safe**
Strengthened Post	*Unsafe*	Generally Unsafe	Generally Unsafe	Generally Unsafe

4.3 Formal Analysis of Service Promotion Correctness

In this section, we characterise the proof obligations in Kmelia from the above-defined constructors and we show how to prove them using Atelier B.

1. **Unsafe case:** the specifier is notified immediately that the promotion is not correct without any proof.
2. **Proven safe case:** the specifier's intent is captured by strengthening/weakening operators and the proof obligations are satisfied by construction.
3. **Safe case:** as in the previous case, the specifier's intent is captured, but to be sure that the promotion is really safe, we must prove that the predicates match the intent (proving the strengthening or weakening).
4. **Generally unsafe case:** in this case we must prove that the new predicates satisfy the client contract.

Listing 2. B pattern for service promotion verification

```
MACHINE
   C_CC_origin_promoted
CONSTANTS
   /* ORIGINAL */
    /* C variables */
      C_v1, ...
    /* C variables updated values  */
      C_v1_new, ...
    /* parameters  of origin service */
      o_param1,... , o_result,
   /* PROMOTION */
    /* C variables promoted in CC */
      CC_v1, ...
    /* C variables promoted in CC, updated values */
      CC_v1_new, ...
    /* parameters  of promoted service */
       p_param1, ..., p_result
PROPERTIES
   /* ORIGINAL */
    /* C variables and origin parameters typing */
       C_v1∈T ∧ o_param1∈T ∧ ...
    /* C invariant related to the above C variables and their updated values */
       Inv_C ∧ Inv_C_new...
    /* postcondition of service origin */
       Post_origin ∧ ...
  /* PROMOTION */
   /* C variables promoted in CC and promoted parameters typing */
       CC_v1∈T ∧ p_param1∈T ∧ ...
   /* promotion variable mapping */
       CC_v1=C_v1 ∧ ...
   /* parameter mapping */
       p_result=o_result ∧ ...
   /* precondition of promoted service - hypothese for the client contract */
       Pre_promoted ∧ ...
ASSERTIONS
   /*precondition of the origin service */
      Pre_origin ∧
   /*postcondition of the promoted service */
      Post_promoted
END
```

At this stage the sub-components are assumed to be proven correct. The main idea behind our correctness checking method for component service promotions is to encode a promoted service as a B machine, in such a way that the consistency proof of it establishes the correctness of the promoted service. Practically, we generate the appropriate B specifications which proof obligations correspond to the client contract on the promoted pre/post-conditions. For each promotion of a provided origin to a promoted, we build a B machine as shown in Listing 2. Indeed, for a B machine with properties P_B, invariant I_B and assertions A_B, the B method generates the following proof obligation: $P_B \wedge I_B \Rightarrow A_B$.

The **CONSTANTS** clause of the B machine contains variables C_v1,... of the C base component, parameters o_param1,... of the *origin* service and its result variable o_result. It also contains their promoted version, prefixed by CC_ for variables and p_ for parameters and the result variable of the promoted service. The **PROPERTIES** clause contains typing information for every variable and parameters involved (C_v1∈T ∧ o_param1∈T ∧ ...), mapping predicates for the promoted variables (CC_v1=C_v1 ∧ ...), and the following predicates which are used as axioms:

- the promoted pre-condition Pre_promoted is supposed satisfied by the client,
- the base component invariant Inv_C ∧ Inv_C_new... has already been proven,
- the original post-condition Post_origin is considered satisfied if the original pre-condition holds.

The **ASSERTIONS** clause contains the original pre-condition Pre_origin and the promoted post-condition Post_promoted.

4.4 Experimental Results

This section presents the experimentations led on the example of Section 4.1. Once edited and verified with COSTO, the specification is visited to extract one B machine for each service promotion. First the sub-component correctness is checked independently with the principles given in [3]. Then the B machines are extracted from the Kmelia specifications. For this experimentation the extraction was done manually, but we developed B extraction plugins to prove component and assembly correctness [2].

The B machine addToEntry_Store of Listing 3 is used to prove the correctness of the Store service in StockManager, which results from the promotion of addToEntry from IntDictionary. The machine contents instantiates the pattern of Listing 2. The analysis of this machine generated 6 proof obligations which were all discharged by the AtelierB prover in the *Automatic force (1)* mode. Additionally, in order to better illustrate our approach, we intentionally introduced the following errors in the promotion contracts:

- The pre-condition of promoted service was weakened by deleting the predicate pid >= 10 from the original one. In this case, regardless of the post-condition there are still proof obligations that are not discharged by AtelierB prover. This corresponds to the column "Unsafe" in Table 1.

Listing 3. StockManager_addToEntry_store extracted B machine

```
MACHINE
    StockManager_addToEntry_store
CONCRETE_CONSTANTS
    /* origin variables */
        o_keys ,
        o_values ,
        o_result ,
        o_noValue ,
    /* promoted variables */
        p_catalog ,
        p_labels ,
        p_stock ,
        p_result ,
    /* origin parameters */
        o_key ,
        o_value ,
    /* promoted parameters */
        p_id ,
        p_qty ,
    /* updated variables*/
        new_o_values ,
        new_p_stock
PROPERTIES
    /* origin typing */
        o_keys ⊆ 1 .. 100 ∧
        o_values ∈ 1 .. 100 ⟶ INT ∧
        o_result ∈ INT ∧
        o_noValue ∈ INT ∧
        o_noValue = - 1 ∧
    /* promoted typing */
        p_catalog ⊆ 1 .. 100 ∧
        p_labels ∈ 1 .. 100 ⟶ INT ∧
        p_stock ∈ 1 .. 100 ⟶ INT ∧
        p_result ∈ INT ∧
    /* origin parameters typing */
        o_key ∈ INT ∧
        o_value ∈ INT ∧
    /* promoted parameters typing */
        p_id ∈ INT ∧
        p_qty ∈ INT ∧
    /* updates variables typing */
        new_o_values ∈ 1 .. 100 ⟶ INT ∧
        new_p_stock ∈ 1 .. 100 ⟶ INT ∧
    /* origin invariant already proved */
        card ( o_keys ) ≤ 100 ∧
    /* origin post */
        o_result = o_key ∧
        new_o_values(o_key) = o_values(o_key)+
                                 o_value ∧
    /* maping predicat */
        p_result = o_result ∧
        p_catalog = o_keys ∧
        p_stock = o_values ∧
        p_id = o_key ∧
        o_value = p_qty ∧
        new_o_values = new_p_stock ∧
    /* pre promoted */
        ( p_id ∈ p_catalog ∧ p_stock ( p_id ) +
               p_qty ≥ 0 ∧ p_qty > 0 )

ASSERTIONS
    /*pre origin*/
        o_key ∈ o_keys ∧ o_values ( o_key ) +
                 o_value ≥ 0 ∧
    /*promoted post*/
        p_result = p_id ∧
        p_catalog = p_catalog ∪ { p_id } ∧
        new_p_stock(p_id) = p_stock(p_id)+p_qty ∧
        new_p_stock ( p_id ) > p_stock ( p_id )
END
```

– Keeping the pre-condition unchanged, we have not introduced other restrictions in the invariant of the composite (restriction by context). The AtelierB could not prove the strengthened post-condition of the promoted service because we have no guarantee that $\mathsf{pqty} > 0$ establishes the predicate $\mathsf{pstock[pid]} < \mathsf{old(pstock)[pid]}$. This is the first case of "Generally Unsafe" in Table 1. This simple case could be detected without launching the prover.
– Strengthening both the pre-condition with $\mathsf{pqty} > 0$ and the post-condition with $\mathsf{pstock[pid]} < \mathsf{old(pstock)[pid]}$ let the new post-condition unsatisfied. Hence the AtelierB proof could not succeed.

The contract expressed by the B pattern of Listing 2 corresponds to the general case of the client service contract for provided services. It contains the proof of pre-condition strengthening. However, the safety property can be proven without consistency of the designer declarations: a weakened post-condition could have been introduced using a STRONGER keyword. This case and the reverse one (less likely to have ensured the safety proof) requires an additional proof.

5 Encapsulation and Observability Rules

In this section we study the impact of state observability on promotion safety. State observability in component models is achieved using either observable variables, accessing methods or attribute controllers. Abstraction and encapsulation are crucial to the scalability of the component approach. To an outside observer, a composite component should not be distinguished from a primitive component and should not be overly complex. Promoting all the services and variables of its sub-components would run contrary to this goal. When deciding what is observable or what is promoted, the designer makes a trade-off between encapsulation and a precise state description. From the verification point of view, the goal is to achieve both abstraction and capitalisation from previous proofs. In the following we note V^O the **observable** subset of the state variables V.

5.1 Observability of Predicates

Predicates containing non-observable state variables are of no use to potential clients of the component. Consequently, we distinguish between observable predicates which contain only variables in V^O from non-observable predicates which can take their variables in V. The observable predicates Inv^O and the non-observable predicates Inv^{NO} form a partition of the invariant Inv. The decision to make a predicate observable is subject to the following guideline (gl) and two rules (r1, r2):

gl. To get observable predicates as meaningful as possible, there must be few (preferably none) non-observable predicates depending only on observable variables.

r1. The pre-condition of a provided service must not contain non-observable predicates because it would make for an unfair contract: the client would have no way to know if the pre-condition is satisfied.

r2. Conversely, the post-condition of a required service, which expresses what is expected, cannot contain non-observable predicates.

More details about the verification of invariants and well-formedness of predicates w.r.t. observability can be found in [3,2].

5.2 Variable Abstraction and Promotion

In Kmelia, one can define a calculated variable (*e.g.* defining $isEmpty$ as $size = 0$). This abstraction mechanism can be used to simplify predicates. If the initial variable is no longer present in observable predicates, it can be safely removed from V^O. This abstraction mechanism is even more useful when building a composite component, since only a subset of the predicates are actually used.

The composite invariant and the pre/post-conditions of its services might depend on the observable variables of its sub-components. In order to preserve encapsulation, these variables have to be seen as variables of the composite (otherwise it would expose the sub-components). In Kmelia the operation of *variable promotion* allows one to define special composite variables that link to observable variables of sub-components. An observable variable vo from a sub-component $c : C$ can be promoted as a variable vp of a composite component (the syntax for that is: `vp FROM c.vo`). The promoted variables retain their types and are abstractions of the read-only access to the vo in their effective contexts. This guarantees the encapsulation principle. The consequences of the observability rules for the provided services are:

– Observable variables in Pre_{origin} must be observable by the clients of *promoted*; removing them in the predicate or marking them as non observable would be equivalent as a weakening of the pre-condition, which is unsafe and forbidden.
– Observable variables in $Post_{origin}$ can optionally be made observable in the composite. If they are not, it can be a weakening of the post-condition in the spirit of information hiding.

5.3 Invariant Promotion

In order to qualify correctly the promoted variables, the corresponding observable invariants are to be promoted too. If these predicates make use of non-promoted variables, the designer will face again a trade-off between abstraction (dropping the predicates) and precise qualification of the variables (promoting the variables needed). In the following we call *reachable properties* of a promoted variable the subset of the predicates in the observable invariant of its origin that depends only on promoted variables. The promoted invariant is usually a weakened version of the original one when all variables are not promoted. However, constraints in the initialisation of the sub-components can strengthen it.

5.4 Consequences on the Promotion Process

The rules governing observability and promotion dependencies are quite simple and easy to integrate in a semi-automated process sketched as follows:

1. determining the sub-components of the composite;
2. selecting the services to promote, and automatically promote the variables needed and their reachable properties:
   (a) optionally modifying the services pre/post-conditions,
   (b) updating needed variables according to the new predicates;
   (c) automatically checking the proof feasibility if applicable (see table 3),
3. optionally selecting additional promoted variables and recompute their reachable properties;
4. optionally defining calculated variables and abstracting existing predicates;
5. assigning variables, services and predicate observability according to the rules previously defined through automatic guidance and verification;
6. checking the proof feasibility and generating proof obligations if applicable using the method described in Section 4.

This iterative promotion process stresses the use of encapsulation and abstraction. The verification method remains unchanged but the extraction process takes the selection of predicates and variables into account. The automatic computation of reachable properties and the observability rules guide the designer in establishing the trade-off between encapsulation and a precise state description before proving the promotion correctness.

6 Related Work

The work presented in this article covers a series of topics including aggregation, composition, promotion, sharing, and contracts in the area of components and services. We compared functional contracts and verification with formal methods in [2]. In the following we focus on the contract based approaches.

Contracts are helpful to deliver rich trusted information. In [18] functional and extra-functional contracts (including dynamic behaviours) provide trust-by-contract components. However the proof of the contracts is not treated at the design level. A component contract classification is proposed by Beugnard et al. [6]. Four levels are considered: *syntactic contracts (i)*, semantic constraints such as *behavioural contracts (ii)* and *synchronisation contracts (iii)* are encountered in various component models; and finally *quality of service (iv)* which is often used at runtime. Applied to services, the classification of Brogi [7] considers also four contract levels: signature, quality of service, ontology for data and protocols. In summary, the word "contract" or "behavioural contract" is often overloaded. We distinguish functional contracts from synchronisation contracts. In the former *e.g.* [17,14], the pre/post-conditions are interpreted in terms of call sequences and (observational) equivalence rather than in logical predicates. The latter category is related to the definition of Meyer's contracts and subcontracts, it is the subject of the remaining of the section.

ConFract [10] inherits from Fractal's interface delegation (promotion) support. ConFract contracts are mainly used for composite components and they are located in Fractal membranes. ConFract contracts are independent entities which are associated to several participants of the composite (external and internal views), and support a rely/guarantee mechanism. ConFract uses the executable assertions language CCL-J to express specifications at interface and component levels. In the case of CCL-J, when a method is called on an interface, the contract controller is then notified and it applies the checking rules. Pre-conditions, post-conditions and method invariants of all contracts "are checked at runtime". A ConFract contract may cover several interfaces and it does not handle promotion, *i.e.* the interface may change by promotion. In Kmelia the promotion contract must conform to the original one and therefore we reuse the proof efforts made on sub-components.

The Service Component Architecture [12,15] is a set of specifications which describes a model for building applications using a Service-Oriented Architecture. The approach, as formalised in [12] is very similar to Kmelia when ports are services but contracts are not handled. A promotion mechanism is proposed to make services visible at the composite level but no transformation is permitted.

Hidden dependencies [13] is another formalism close to Kmelia: it supports contracts for provided and required services and it handles four types of service dependencies. But unlike Kmelia the dynamic compatibility is weak and services are flat operations supporting sub-typing. Promotion is defined by delegation dependencies, the promoted services are said to be visible while the original ones are said to be hidden. A fixed-point equation solves the dependency and contract rules. The contents of the dependency relation is not given, and promotion links seem to be governed by an $\wedge$-rule rather than an $\Rightarrow$-rule. No verification support is provided yet.

7 Conclusion

In this article we presented the issue of the correct promotion of services in component models. Our approach contributes at the level of correct-by-construction composite components and also at the level of the consistency of component assemblies. We described several kinds of promotions, their usefulness and the conditions of their safety. We defined operators on predicates to be used during the promotion of a service to make explicit the intent of the designer and reduce the proof effort. In particular, using these operators allows to easily rule out promotion kinds in systematically unsafe situations, and automatically accept always safe promotions kinds, as well as defining simple heuristics to warn the designer about generally unsafe situations.

We presented and illustrated a verification method that capitalises from previously proven properties [3]. This method is based on the generation of B models from relevant parts of the Kmelia specifications and their analysis using B tools. We then described a model and a process that support scalability through abstraction while taking into account the verification needs.

The COSTO tool currently determines the always safe and always unsafe cases according to the keywords involved, but we aim at adding wizards to assist the specifier in following the process sketched in Section 5 and define simple heuristics for determining when calculated variables would favour abstraction. Other short term perspectives of this work include its full implementation in COSTO with a new B extraction plugin integrating the promotion language features introduced in this article, as well as a feedback analysis. A medium term perspective is to extend the current primitives to behavioural contract promotion, as a follow-up of the work on the behavioural compatibility rules already defined in Kmelia and behavioural refinement of [5].

Note: A separate appendix for this article is available at:
`http://www.lina.sciences.univ-nantes.fr/coloss/download/facs10_app.pdf`

References

1. Abrial, J.R.: The B-Book Assigning Programs to Meanings. Cambridge University Press, Cambridge (1996) ISBN 0-521-49619-5
2. André, P., Ardourel, G., Attiogbé, C., Lanoix, A.: Contract-based Verification of Kmelia Component Assemblies using Event-B. In: Proceedings of FESCA (2010)

3. André, P., Ardourel, G., Attiogbé, C., Lanoix, A.: Using assertions to enhance the correctness of kmelia components and their assemblies. ENTCS 263, 5–30 (2010); Proceedings of FACS 2009
4. Baude, F., Caromel, D., Henrio, L., Morel, M.: Collective interfaces for distributed components. In: CCGRID, pp. 599–610. IEEE Computer Society, Los Alamitos (2007)
5. Bauer, S.S., Hennicker, R., Bidoit, M.: A Modal Interface Theory with Data Constraints. In: Proceedings of the 13th Brazilian Symposium on Formal Methods (SBMF 2010). LNCS series of Springer (to appear)
6. Beugnard, A., Jézéquel, J.M., Plouzeau, N., Watkins, D.: Making components contract aware. Computer 32(7), 38–45 (1999)
7. Brogi, A.: On the Potential Advantages of Exploiting Behavioural Information for Contract-based Service Discovery and Composition. Journal of Logic and Algebraic Programming (March 2010), `http://dx.doi.org/10.1016/j.jlap.2010.01.001`
8. Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., Stefani, J.B.: The Fractal Component Model and Its Support in Java. Software Practice and Experience 36(11-12) (2006)
9. Bures, T., Hnetynka, P., Plasil, F.: Sofa 2.0: Balancing advanced features in a hierarchical component model. In: Proceedings of SERA, pp. 40–48. IEEE Computer Society, Los Alamitos (2006)
10. Collet, P., Malenfant, J., Ozanne, A., Rivierre, N.: Composite Contract Enforcement in Hierarchical Component Systems. In: Lumpe, M., Vanderperren, W. (eds.) SC 2007. LNCS, vol. 4829, pp. 18–33. Springer, Heidelberg (2007)
11. Crnkovic, I., Larsson, M.: Component based software engineering - state of the art. Technical Report ISSN 1404-3041 ISRN MDH-MRTC-15/2000-1-SE, Mälardalen University (January 2000)
12. Ding, Z., Chen, Z., Liu, J.: A rigorous model of service component architecture. Electr. Notes Theor. Comput. Sci. 207, 33–48 (2008)
13. Enselme, D., Florin, G., Legond-Aubry, F.: Design by contract: analysis of hidden dependencies in component based application. Journal of Object Technology 3(4), 23–45 (2004)
14. Graf, S., Quinton, S.: Contracts for bip: Hierarchical interaction models for compositional verification. In: Derrick, J., Vain, J. (eds.) FORTE 2007. LNCS, vol. 4574, pp. 1–18. Springer, Heidelberg (2007)
15. Krämer, B.J.: Component meets service: what does the mongrel look like? ISSE 4(4), 385–394 (2008)
16. Medvidovic, N., Taylor, R.N.: A Classification and Comparison Framework for Software Architecture Description Languages. IEEE Transactions on Software Engineering 26(1), 70–93 (2000)
17. Reussner, R., Poernomo, I., Schmidt, H.W.: Reasoning about Software Architectures with Contractually Specified Components. In: Cechich, A., Piattini, M., Vallecillo, A. (eds.) Component-Based Software Quality. LNCS, vol. 2693, pp. 287–325. Springer, Heidelberg (2003)
18. Schmidt, H.: Trustworthy components-compositionality and prediction. J. Syst. Softw. 65(3), 215–225 (2003)

Systems-Theoretic View of Component-Based Software Development*

Daniel Côté, Michel Embe Jiague, and Richard St-Denis

Département d'informatique
Université de Sherbrooke
Sherbrooke (Québec), J1K 2R1, Canada
{Daniel.R.Cote,Michel.Embe.Jiague,
Richard.St-Denis}@USherbrooke.ca

Abstract. This paper investigates component-based software development in the perspective of systems theory. In the proposed systems-theoretic view, a complex system is organized hierarchically from horizontal and vertical aggregations of components, but more important is the explicit control at each level of the hierarchy. Control actions are then determined by controllers that enforce constraints imposed on components and their interaction, and thus reduce their degree of autonomy. Not only the system behavior is restrained but nonfunctional properties emerge at each level. The finer the exercised control, the richer emergent properties should be. Therefore, achieving nonfunctional properties, such as liveness, predictability, safety and security, corresponds to solving control problems. The supervisory control theory initiated in the early and mid eighties is a mathematical apparatus that helps to accomplish this task in a rigorous way.

Keywords: Systems theory, supervisory control theory, component-based software development, component model, formal method, hierarchical control, controller synthesis.

1 Introduction

According to Szyperski's definition [29], a software component is a delineated unit of composition that is contractually bound to carry out useful functions through clearly defined interfaces. One important element missing in this definition is the notion of control, which appears only occasionally in component models under an explicit form. For instance, in the Fractal component model a component provides an open set of control capabilities [5]. The implementation part of a component is composed of other components, called sub-components, which are under the supervision of the controller of the enclosing component. The controller takes actions regarding the behavior to be associated with a component. In particular, it can superpose control behavior to the behavior of its sub-components. Unfortunately, the way these controllers are designed is not supported by a suitable theoretical framework. Most recent formal models of interacting components pay little attention to aspects related to control. In the theory elaborated by

* The research described in this paper was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC).

Broy [6], elements associated with control are stated in assumption/commitment specifications for services. Such specifications ensure that input sequences within a service domain lead to a well controlled behavior. In the formal model defined by Chen *et al* [8], control is limited to flow of control inside a process (glue code) on when to call out or wait for a call to its provided services. The rCOS method suggests to model the flow of control and synchronization of a component by a state machine diagram which represents the reactive behavior of the component [15]. An interesting feature concerns the verification of the dynamic consistency of the sequence diagram and state machine diagram. They are translated into CSP processes to check deadlock-freedom.

The proposal defended in this paper was deeply influenced by Nancy Leveson's work on safety engineering in the perspective of modern systems thinking and systems theory [20], but with the aim to put forward the idea that a mathematically-based control theory can contribute to a general theoretical framework for component-based software engineering (CBSE):

> "in systems theory, emergent properties, such as safety, arise from the interactions among the system components. The emergent properties are controlled by imposing constraints on the behavior of and interactions among the components. Safety then becomes a control problem where the goal of the control is to enforce the safety constraints [20]."

In this perspective, solving control problems is considered to be essential and this task can be realized by taking advantage of the *supervisory control theory* (SCT) [26]. This theory peculiar to discrete event systems (DES) tackles control problems with respect to various structural configurations, in particular hierarchical control variants suitable to manage complexity as recognized by systems theory. It puts the focus on formulation of conditions for the solvability of different control problems and design of synthesis procedures that automatically derive controllers, which are correct by construction. Therefore, CBSE could benefit from SCT, especially when considering dynamic adaptive components and dynamic reconfigurations. The former include a special component that delegates control to the other components while keeping the control over them itself in order to guide structural changes in components and their coordination [1]. The latter require a monitor not only for interpreting a configuration, but also for acting on architectural elements to reconfigure a system in a consistent way [10]. All these forms of control raise many open issues that could be solved by considering the mutual advantages between CBSE and SCT.

Several component models, in particular the one of rCOS, adopt the principle of *separation of concerns*. This paper focuses on one concern, namely the *control*. It shows how a hierarchical control variant of SCT can be adapted as a theoretical foundation for this concern. The ultimate goal is to separate the control from the interface and implementation, as proposed in the Fractal component model, by bringing to the fore a design process based on the formulation and resolution of a specific control problem, including the synthesis of controllers. The main advantage of this approach is to avoid negative impacts such as overlapping of functionality, dissipation of an important concern (control) into several architectural elements or combination of code specific to a concern with those that implement other concerns.

Since synthesis problems and verification problems are dual, the approach advocated in this paper is in some sense the converse of the one proposed by the Sifakis's team in which components (including possible controllers) are specified in a subset of the BIP language and verification techniques combined with heuristics are used to check deadlock-freedom [3]. There is, however, more information in the latter, since controllers missing in the former must be calculated. This calculation does not require heuristics to verify deadlock-freedom (or other types of properties) because of a less expressive formalism to specify the behavior of components. The system is deadlock free by construction. Nevertheless, a more expressive formalism can be used, but heuristics or human intervention (when heuristics fail) are required during the execution of the synthesis procedure [28]. The main contributions of this paper are the following.

– It shows how to match the concepts of a SCT hierarchical control variant [32] with those of a general component model. This is not at first sight obvious considering the richness of the theory with respect to the number of hierarchical control variants. For instance, in the aggregate multilevel hierarchy approach proposed by Leduc *et al* [18], hierarchical system construction with an arbitrary number of layers is not really supported since the abstraction process cannot be repeated along the vertical line. It is limited to two layers [19]. The two-tiered architecture as defined in this framework does not yield another generic model, including the interface, unless a composition mechanism is applied in an ad hoc manner. In the structural multilevel hierarchy approach developed by Ma and Wonham [22], the encapsulation is open because the internal transition structure is accessible to the synthesis procedure and the unique optimal nonblocking supervisor calculated by a synthesis procedure is directed against the whole transition structure.
– It provides abstraction rules to endow the interface part of a component with a standard control technology in order to achieve conditions that guarantee safety, deadlock-freedom and progress by construction. This set of abstraction rules ends up being more restrictive than what it is suggested by Wong and Wonham [32], which requires the output control consistent property (OCC) in order to obtain the finest possible control granularity. In the proposed definition the OCC requirement is discarded with the effect of the relinquishment of control options in the implementation part of a component. Generally, the OCC property leads to more complex models for the implementation part. Thus, it represents a difficulty for practitioners.
– It introduces component properties, considers horizontal composition and vertical composition as well as superposition of control, and it shows that these properties are invariant under these three operations. This allows for their combination in an arbitrary way with an unrestrained number of components and an unrestrained number of abstraction levels. Thus, the proposed approach satisfies an important requirement mentioned in a survey on software component models [17], namely theories must support systematic composition and infer properties about the result of applying a composition operator to components. In other words, it preserves the formal guaranties provided by SCT throughout the entire design process. It is in the same spirit of some work in the domain [11] (flexible composition preserves deadlock-freedom), [6] (separate refinements of the components of a system lead to a refinement of the composed system), [16] (the characteristics of encapsulation

and compositionality are preserved) and [8] (the disjoint union of closed components forms another closed component).

The rest of this paper is divided into substantial sections. Section 2 introduces a running example in order to make readable the next sections to readers less familiar with a control-theoretic approach. Section 3 presents basic notions of SCT in order to grasp the nature of a typical control problem when stated formally. Most notions are illustrated with the running example. Section 4 outlines the hierarchical control in its more abstract form around a key theorem, while establishing a correspondence between concepts of SCT and those of a general component model. It also provides a useful method to apply it. Section 5 details three basic constructs of components used to assemble components horizontally and vertically with control while satisfying the conditions of the theorem introduced in the previous section. A conclusion situates this work in a technical discussion about its advantages, contrasting it with Gössler and Sifakis'work, and mentions how it can evolve in the future with respect to a list of open issues.

2 A Running Example: Composition and Control of BPEL Processes

To illustrate component-based software development in the paradigm of systems theory within the limits of the supervisory control theory as suggested in this paper, a concise example is provided. In this example components are Web services implemented as BPEL[1] processes. The example is built around an on-line store that places at its customers' disposal a Web application from which they can purchase goods. In order to maintain a reasonable stock, goods are acquired from manufacturers, then conveyed and stored in a stockroom. Deliveries to customers are worldwide and made by using a service supplied by a partner shipping company. When required (e.g., due to non-shipping facilities in a target country), the shipping company may exploit other shipping services to reach customers. The Web application is a front end of a workflow implemented as a BPEL process, called `Shop`. The latter interacts with other BPEL processes: `Bank` for credit authorization and debit, `Manufacturer` for goods production and `Ship` for delivery scheduling. The process is such that an end user of the Web application can review its order before submitting it to final processing.

Figure 1 shows all the BPEL processes in a schematic notation. The reader can presume that the auxiliary BPEL processes have more complex WSDL interfaces, but only a part of the whole is effectively used by the process `Shop`. The interpretation of the BPEL processes is straightforward, except at the point where `Shop` interacts with `Bank` through the messages $\texttt{validate(creditCard)}_{\texttt{i}}$ and $\texttt{validate(creditCard)}_{\texttt{o}}$. After receiving the response from the bank, `Shop` may report an error, because the customer is not creditworthy, or complete the order. In the last case the control flow is divided into three parallel threads of execution that perform activities towards the goal of confirming the customer order. The messages $\texttt{getProdQty}_{\texttt{i}}$ and $\texttt{getProdQty}_{\texttt{o}}$ have no counterpart in the auxiliary processes. They are internal messages. BPEL processes hold

[1] BPEL stands for *Business Process Execution Language*. It is an OASIS standard executable language for specifying actions within business processes with Web services.

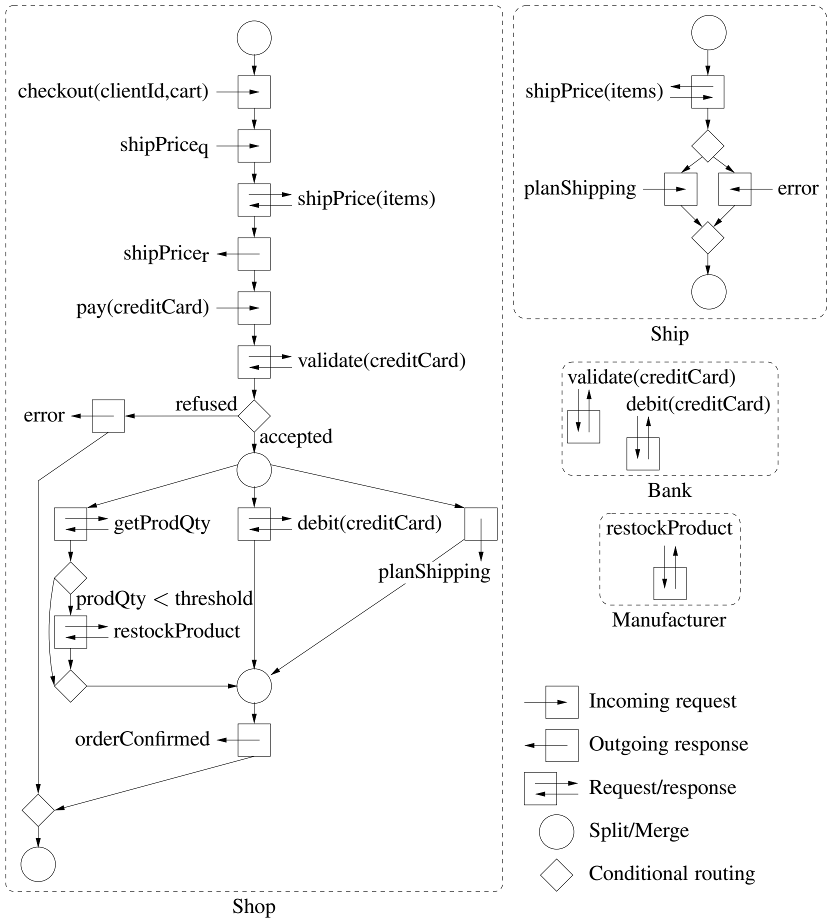


Fig. 1. Workflows of BPEL processes

information associated to each of its threads. For instance, when `Ship` receives the request `shipPrice(items)`$_i$, information about items to ship and destination[2] is kept until an error occurs or the determination of a shipping schedule.

3 The Implementation and Control Parts — Introduction to the Supervisory Control Theory

The supervisory control theory is an established theory in the domain of DES. Since the last 30 years, it has been refined and perfected to contemplate a wide range of problems with various formalisms and analysis tools. There is a wealth of literature on this theory and its application in circumscribed domains. Ramadge and Wonham, the

[2] Due to space limitation some arguments are missing in the messages.

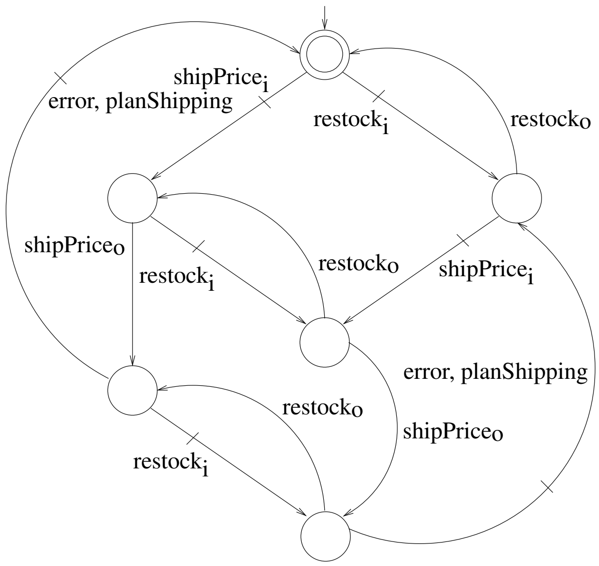


Fig. 2. Possible interactions with the shipping company and the manufacturer

founders of this theory, surveyed language-based formulation of control problems, including their underlying properties such as controllability, observability, nonblocking and nonconflicting, together with their automaton representation for computational and implementation purposes [26]. Instead of using formal languages or automata for describing admissible event trajectories, temporal logics have also been considered in order to specify more complex liveness and real-time properties [2]. Holloway *et al* wrote a survey paper on Petri net methods for controlled DES [12]. In their book, Kumar and Garg developed basic results about control under complete observation, control under partial observation and control of nonterminating behavior using the lattice theory [14]. Cassandras and Lafortune addressed more subjects in their textbook on DES, particularly diagnosis, decentralized control, control of Petri nets and hybrid systems [7]. Finally, Bherer *et al* included in their paper on the control of parameterized DES an overview of techniques to limit the negative impact of the state-space explosion problem shared by most control problems and detailed one of them [4].

In SCT system behavior is mostly modeled by an automaton $G := (X, \Sigma, \delta, x_0, X_m)$, where X is a set of states; Σ is a finite set of events; $\delta : X \times \Sigma \to X$ is the partial transition function; x_0 is the initial state; and X_m is the subset of marked states, which represents the completed tasks. Such a model is often obtained by a synchronous product of pairwise disjoint component models. It is convenient to introduce the extended transition function $\delta : X \times \Sigma^* \to X$ as in the usual way [13]. The *closed behavior* and *marked behavior* of G are defined, respectively, as follows:

$$L(G) := \{w \in \Sigma^* \mid \delta(x_0, w)!\} \text{ and } L_m(G) := \{w \in \Sigma^* \mid \delta(x_0, w) \in X_m\},$$

where $\delta(x, w)!$ is an abbreviation for the expression "$\delta(x, w)$ is defined". The *active event set* of a state x, noted $\Gamma(x)$, is defined by $\{\sigma \mid \delta(x, \sigma)!\}$. For computational reasons, it is generally assumed that a DES has finitely many states. The pair $\langle L(G), L_m(G)\rangle$ is a language model of the system. Figure 2 shows a fragment of the automaton that represents the free behavior of the process `Shop`, more specifically the possible interactions with the shipping company and the manufacturer in its implementation part. It should be noted that transitions cannot be labelled with predicates and update relations as in the BIP framework [3]. However, it is possible to use variables with finite domains and represent their different values by states.

Sometimes it is convenient to make abstraction of the underlying transition structure and consider only an alphabet Σ and a pair $\langle L, L_m\rangle$ with $L, L_m \subseteq \Sigma^*$ and $L_m \subseteq L = \overline{L}$, where $\overline{L}$ denotes the prefix closure of L [14]. These two types of model will be used interchangeably in this paper. Let $\mathcal{F}_L := \{K \in \wp(L) \mid K = \overline{K}\}$, where $\wp(L)$ denotes the power set of L, be the set of closed sublanguages of L. The term $\wp^2(L)$, which is a short form of $\wp(\wp(L))$, represents the set of families of sublanguages of L.

Let x be the current state of G. If $x \notin X_m$ and $\Gamma(x) = \emptyset$, there is a deadlock, since no further event can occur. If x belongs to a strongly connected component that consists solely of unmarked states without outgoing transitions, there is a livelock. If $\overline{L_m(G)} = L(G)$, G is said nonblocking. Intuitively, this means that all subtasks in G can be eventually completed. Otherwise G is blocking (i.e., $\overline{L_m(G)} \not\supseteq L(G)$), a deadlock or a livelock can happen. The *nonblocking* property is predominant in many control problems. In the language-based formulation, the condition for $E \subseteq L$ to be nonblocking is

$$\overline{E} = \overline{\overline{E} \cap L_m}. \tag{1}$$

The *controllability* property is another important concept of SCT. A language $K \subseteq \Sigma^*$ is *controllable* with respect to L, if $\overline{K}\Sigma_u \cap L \subseteq \overline{K}$, where $\Sigma_u \subseteq \Sigma$ is the set of uncontrollable events ($\Sigma_c := \Sigma - \Sigma_u$ is the set of controllable events). In the Web application example, the incoming requests correspond to controllable events (indicated by a slanting bar on transitions in Figure 2) and the outgoing responses correspond to uncontrollable events. Responses to requests are uncontrollable because the initiator of requests has no control on when and if they will eventually arrive. An event that represents a timeout is also uncontrollable. Intuitively, a language K is controllable if any subtask of K followed by an uncontrollable event that is possible in L is also a subtask of K. Let $\Lambda := \wp(\Sigma_c)$ be the set of all *control actions* and $\lambda \in \Lambda$. If $\sigma \in \lambda$, then σ is disabled; otherwise, it is enabled. An uncontrollable event is always enabled. This way of implementing control is called *the standard control technology*. In fact, a control technology can be seen as an implementation of a *control structure*, which is a means to represent control abstractly by a family of controllable sublanguages. Formally, a control structure is a map $\mathcal{C} : \mathcal{F}_L \to \wp^2(L)$ that satisfies four axioms (for every $H \in \mathcal{F}_L$): join closure ($\mathcal{C}(H) \subseteq \wp(H)$ is a complete upper semilattice), nontriviality ($\emptyset, H \in \mathcal{C}(H)$), prefix closure ($K \in \mathcal{C}(H) \implies \overline{K} \in \mathcal{C}(H)$) and inheritance (for $F \in \mathcal{F}_L$ and $H \subseteq F$, $\mathcal{C}(F) \cap \wp(H) \subseteq \mathcal{C}(H)$, with equality if $H \in \overline{\mathcal{C}}(F) := \mathcal{F}_F \cap \mathcal{C}(F)$) [32]. Further restrictions can be imposed on such maps in order to establish associations between control structures and control technologies. A control technology induces a *standard*, *locally definable* control structure. In particular, the *standard* control technology induces a standard, locally definable control structure.

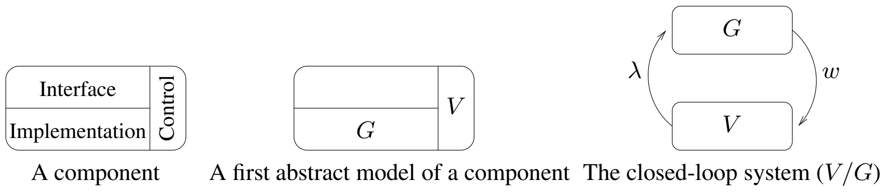

Interface
Implementation
Control
A component
G
V
A first abstract model of a component
G
V
λ
w
The closed-loop system (V/G)

Fig. 3. Control in the framework of the supervisory control theory

Furthermore, a standard, locally definable control structure can always be implemented by a control technology (see [30] for more details).

In SCT, any controlled system is seen as a solution of a control problem, which typically consists in synthesizing a supervisor to restrain the uncontrolled behavior of a DES in order to achieve a given specification. More precisely, given a model $\langle L, L_m \rangle$ of a DES and a specification E, a *nonblocking* supervisor $V : L \to \Lambda$ must be calculated such that the system behavior under the supervision of V is restrained to the greatest possible number of event trajectories of L_m included in E. The language defined by these trajectories is called the *supremal controllable sublanguage* of E with respect to L, since E is not necessarily controllable, and the synthesized supervisor is termed *maximally permissive*. In the running example, the specification prescribes some ordering constraints on the occurrences of messages (e.g., the response `shipPrice(items)`$_\text{O}$ must occur before `planShipping`) or indicates bad states for security reasons. Intuitively, a specification can be interpreted as a model of control glue usually written in a coordination language.

In general, given $F \in \mathcal{F}_L$ and $H \subseteq F$, the map $\kappa_F : \wp(F) \to \mathcal{C}(F)$ assigns to H the supremal controllable sublanguage of H with respect to F. For notational purpose, $\overline{\kappa}_F := \kappa_F|_{\mathcal{F}_F}$ is the restriction of κ_F to $\mathcal{F}_F$. The DES and supervisor are brought together in a *closed-loop system* denoted by V/G. The DES generates events with respect to control actions determined by the supervisor as shown in Figure 3.

Several synthesis algorithms exist to calculate supervisors. They differ substantially in the formalisms used to represent the DES and specification. Some of them integrate BDD to deal with very large systems (e.g., [22]). Thus, the application of SCT to component-based software development is limited by the expressiveness of formalisms supported by those algorithms and the size of models.

4 The Interface Part — The Hierarchical Control Variant

In Figure 3, the interface has been excluded from the abstract model of a component. This omission was intentional because the emphasis was on the control exercised on the implementation. To take into consideration all the constituents of a component, in particular the interface, one can take advantage of hierarchical control, a variant of SCT. It is developed around the schema at the left of Figure 4. Let G_{lo} (or $\langle L, L_m \rangle$) be a model of the implementation and T be the alphabet associated to the interface. A high-level model G_{hi} (or $\langle M, M_m \rangle$ with $M, M_m \subseteq T^*$ and $M_m \subseteq M = \overline{M}$) is

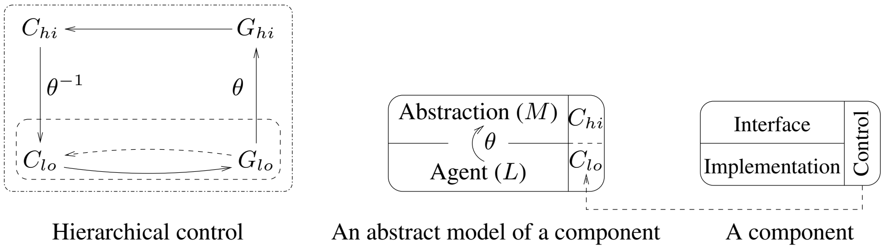


Fig. 4. Elements of hierarchical control

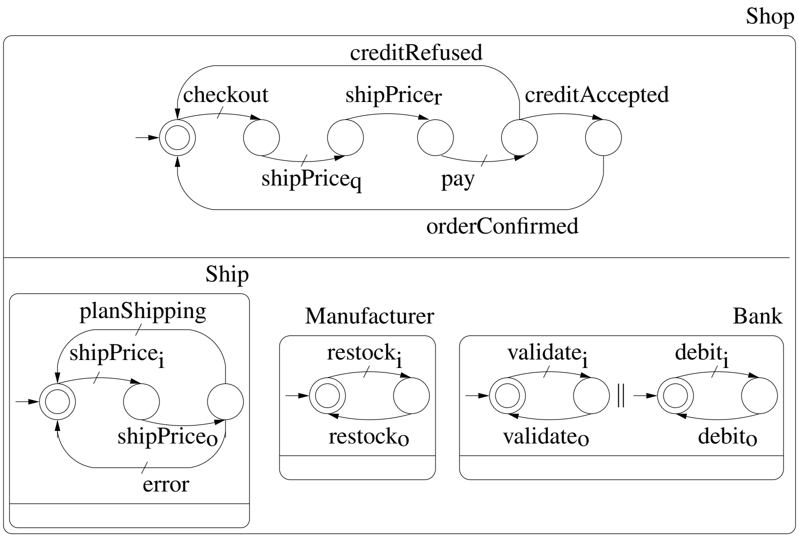


Fig. 5. Hierarchical abstract view of BPEL processes

obtained from the low-level model G_{lo} by using a causal reporter map $\theta : L \to T^*$ (θ is prefix-preserving). Thus, G_{hi} is a model of the interface with $M := \theta(L)$ and $M_m := \theta(L_m)$. Furthermore, θ and $\theta^{-1} : \wp(M) \to \wp(L)$, the inverse image map, represent the information channel and command channel respectively. In other words, G_{hi} is an abstraction of G_{lo}, the agent. The latter is supervised by the control structure $\mathcal{C}_{lo}$, but the control in the upper level, represented by the control structure $\mathcal{C}_{hi}$, is indirect: in a dynamic closed-loop model the control actions of $\mathcal{C}_{hi}$ are dispatched to $\mathcal{C}_{lo}$ through the command channel, the result of this translation is applied to the agent, the abstraction is then driven by the agent via the information channel and the behavior of the abstraction is recorded in $\mathcal{C}_{hi}$ in order to complete the closed loop [32].

Figure 5 depicts the hierarchical structure of components, which is an abstract view of the BPEL processes in Figure 1 or Web services with WSDL files when deployed. The interface exposed by `Shop` results from the use of a specific causal reporter map. Depending on applications, different causal reporter maps lead to different interfaces. At the implementation level, `Shop` uses the interfaces of `Ship`, `Manufacturer` and

Bank. The interface of the latter is outlined as the shuffle product (interleaving) of two simple request/response automata. The automaton modeling the interface of Ship indicates that after generating the controllable event $\texttt{shipPrice}_\texttt{i}$, the corresponding response $\texttt{shipPrice}_\texttt{o}$ can occur. The automaton can then be reset following the occurrence of the controllable events error or planShipping. The possible interactions between Shop and the auxiliary processes are then represented by an automaton with 24 states for which a fragment has been given in Figure 2. To obtain the appropriate interaction in the implementation of Shop, a specification must be provided in order to calculate a controller. In this example the specification corresponds to the behavior of the BPEL process Shop (as given at the left of Figure 1) and it is controllable. In fact, the specification is most probably an over specification which is the customary way of programming. For more complex situations involving parallelism, the human brain cannot encompass the whole structure. In this case, the synthesis presents many advantages.

Let $L_{voc} := \{\epsilon\} \cup \omega^{-1}(T) \subseteq L$ where $\omega : L \rightarrow T \cup \{\epsilon\}$, called the tail map of θ, is defined as follows ($s \in \Sigma^*$ and $\sigma \in \Sigma$): $\omega(\epsilon) := \epsilon$ and

$$\omega(s\sigma) := \begin{cases} \tau, \text{ if } \theta(s\sigma) = \theta(s)\tau; \\ \epsilon \ \text{ if } \theta(s\sigma) = \theta(s). \end{cases}$$

The set L_{voc} is the set of vocal strings, which are the strings of L that cause the generation of an event through θ [32]. The causal reporter map θ can then be expressed as follows:

$$\theta(\epsilon) = \epsilon$$

$$\theta(s\sigma) = \begin{cases} \theta(s)\omega(s\sigma), & \text{if } \omega(s\sigma) = \tau; \\ \theta(s) & \text{otherwise.} \end{cases}$$

The central theorem of the hierarchical control variant gives conditions for preserving the nonblocking property as defined by Equation 1.

Theorem 4.1. *(theorem 6 in [32]) Let $\mathcal{C}_{lo}$ be a standard control structure on L and $\theta : L \rightarrow T^*$ be a causal reporter map. Suppose that*

$\mathcal{C}_{hi}(M) = \theta(\mathcal{C}_{lo}(L))$ *(control consistency),*

$\theta^{-1}(M_m) = L_m$ *(consistency of marking),*

θ is an observer, and

$\theta_v^{-1} \circ \overline{\kappa}_M \leq \kappa_L \circ \theta^{-1} \circ \overline{\kappa}_M$ *(partner-freedom).*

Then, for all $E \subseteq M$, $\kappa_M(E)$ is nonblocking $\Leftrightarrow$ $\kappa_L \circ \theta^{-1}(E)$ is nonblocking.

The first condition is a specific case of the control consistency property, in which $H = L$, as illustrated in the left part of Figure 6. It states that the projection of any language that belongs to $\mathcal{C}_{lo}$ is controllable in the upper level. In particular, this property ensures that $\mathcal{C}_{hi}$ is a control structure on M [32].

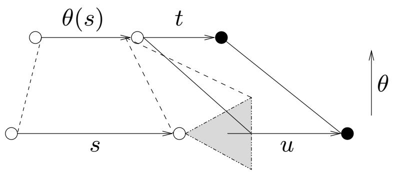
The second condition can be easily satisfied by defining $L'_m = \theta^{-1}(M_m)$ (generally $\theta^{-1}(M_m) \supseteq L_m$) and substituting L'_m for L_m. This artifice is purely theoretical. In practice, this pseudo expansion of L_m implies that a complete task in G_{hi} is not promptly completed in G_{lo}.

The third condition requires that θ be an observer. This property can be characterized as follows (lemma 2.1 in [33]). Let $H \subseteq L$. Then θ is an H-observer *iff*

$$(\forall s \in L)(\forall t \in T^*)\theta(s)t \in \theta(H) \implies (\exists u \in \Sigma^*)su \in H \wedge \theta(su) = \theta(s)t.$$

$$\begin{tikzcd}
\mathcal{F}_M \arrow[r, "\mathcal{C}_{hi}"] & \wp^2(M) \\
\mathcal{F}_L \arrow[u, "\theta"] \arrow[r, "\mathcal{C}_{lo}"'] & \wp^2(L) \arrow[u, "\theta"']
\end{tikzcd}$$

$(\forall H \in \mathcal{F}_L)\; \mathcal{C}_{hi}(\theta(H))=\theta(\mathcal{C}_{lo}(H))$



$\theta(s)t \in \theta(H) \Longrightarrow (\exists u \in \Sigma^*) su \in H \wedge \theta(su)=\theta(s)t$

Fig. 6. Control consistency of $(L, \theta, \mathcal{C}_{lo})$ [32] and characterization of the observer property

Intuitively, if a given behavior of the abstraction can be extended to an admissible string, no matter where the agent is, as long as its image is $\theta(s)$, its behavior can be extended to a string that belongs to H with $\theta(su) = \theta(s)t$ (see the illustration in the right part of Figure 6). If $H = L, \theta$ is an observer. The observer property is closely related to the concept of observation equivalence defined by Milner [24]. Following Wong and Wonham, a causal reporter map $\theta : L \dashrightarrow T^*$ is an observer *iff* the transition structure obtained by relabeling G_{lo}, the automaton for $\langle L, L_m\rangle$, with events in T according to the output behavior of θ, can be reduced to a deterministic quotient which is a bisimulation equivalence of the relabeled automaton containing no unobservable transitions. Thus, given an arbitrary causal reporter map, Wong and Wonham's algorithm [33] modifies this map, adding further vocalized transitions, until the coarsest observer which is finer than the given map is obtained. This new (observer) map can then be used as is. But it usually serves as a heuristic concerning the modifications that should be made to the original map in order to make it into an observer. The interface exhibited by `Shop` in Figure 5 is in fact an observer. It should be noted that the event `creditAccepted` should not normally appear in the interface because it corresponds to an internal message. However, it has been added to the interface because the original causal reporter map was not an observer. Also, the message `error` has been renamed to `creditRefused` in the interface (via the causal reporter map) in order to maintain uniformity.

In the last condition, the expression $\kappa_L \circ \theta^{-1} \circ \overline{\kappa}_M$ is interpreted as the low-level implementation of the high-level synthesis. The expression $\theta_v^{-1} \circ \overline{\kappa}_M$ refers to all the vocal strings corresponding to the high level synthesis (the term θ_v is the restriction of θ to the set of vocal strings, i.e., $\theta_v := \theta|_{L_{voc}}$). Therefore, this condition means that the implementation in the lower level captures all the vocal strings corresponding to the high level synthesis.

Unfortunately, applying Theorem 4.1 constitutes a serious challenge for practitioners in the design of real component systems. Therefore, this theorem has been adapted to a weaker form of control consistency, namely $\mathcal{C}_{hi}(M) \subseteq \theta(\mathcal{C}_{lo}(L))$, when the specification for the abstraction is controllable [9]. Under this restriction, $E \subseteq M$ is replaced by $\kappa_M(E) \in \mathcal{C}_{hi}(M)$ in the conclusion of Theorem 4.1, that is,

$$\kappa_M(E) \text{ is nonblocking} \Leftrightarrow \kappa_L \circ \theta^{-1}(\kappa_M(E)) \text{ is nonblocking},$$

since κ_M is idempotent. Compared with strict control consistency, the weak form gives access to fewer control options in $\mathcal{C}_{lo}$ by the upper level, thus control becomes coarser as the hierarchy of abstraction builds up. Nevertheless, it often reveals adequate for practical use as it embodies the usual trade-off made to obtain coarser-grained interfaces (i.e., simpler abstractions) in order to encapsulate complexity.

Generally, the low level fits with the standard control technology ($\Sigma = \Sigma_c \cup \Sigma_u$), which induces a standard, locally definable control structure $\mathcal{C}_{lo}$ on L. The goal is to equip the abstraction with the standard control technology. This leads to control coincidence, $\theta^{-1}(\mathcal{C}_{hi}(M)) \subseteq \mathcal{C}_{lo}(L)$, which means that any controllable language in the upper level matches a controllable language in the lower level through the inverse image map. Finally, it can be shown that the weak control consistency and partner-freedom properties are fulfilled under the condition that the control coincidence property holds [9]. Clearly θ must be defined so that $\{T_c, T_u\}$ is a partition of T. Assume the following definitions:

$$X_\tau := \{s\sigma \in L_{voc} \mid \omega(s\sigma) = \tau\};\ \ T_\theta := \{\tau \in T \mid X_\tau \neq \emptyset\};$$
$$T_c := \{\tau \in T_\theta \mid X_\tau \subseteq \Sigma^* \Sigma_c\};\ \text{ and }\ T_u := T_\theta - T_c. \tag{2}$$

Without loss of generality, T can be considered equal to T_θ. An event τ is controllable ($\tau \in T_c$) if all vocal strings that cause the generation of τ end with a controllable event in Σ_c. It can be observed that $\theta_v^{-1}(t\tau) := \theta^{-1}(t\tau) \cap L_{voc} \subseteq \Sigma^*\Sigma_c$ and $\emptyset \neq \omega^{-1}(\tau) \subseteq \Sigma^*\Sigma_c$ for all $t\tau \in M$ and $\tau \in T_c$. This is the clue to establish the control coincidence property.

The results presented in this section highlight the need to coordinate correctly aggregation of components.

5 Composition of Components

There are three cases to consider when composing components: i) components are aggregated horizontally without adding supplementary control; ii) components are aggregated vertically without adding supplementary control; and iii) supplementary control is superposed to the control of the abstraction of a component. According to the weak form of Theorem 4.1, weak control consistency, consistency of marking, observer, and partner-freedom properties must be preserved in all these cases in order to combine such operations in an arbitrary way with an unrestrained number of components since the underlying operators (composition of functions and interleaving) are associative.

5.1 Horizontal Aggregation without Additional Control

The case in which components are pairwise disjoint is considered ($\Sigma_1 \cap \Sigma_2 = \emptyset$). In the context of reusable components, this limitation imposed on components is reasonable because if instances share events or if there is any other form of dependency between them, instantiation becomes tricky. Thus, the interleaving operator ("$\|$") between languages (or families of languages) [14] is used in the horizontal composition as shown in Figure 7. It is also assumed that the control structures are standard and locally definable, since they are induced by the standard control technology. The following basic properties hold:

$$\theta_1(H_1) \parallel \theta_2(H_2) = \theta(H_1 \parallel H_2),\ \theta_1^{-1}(N_1) \parallel \theta_2^{-1}(N_2) = \theta^{-1}(N_1 \parallel N_2).$$

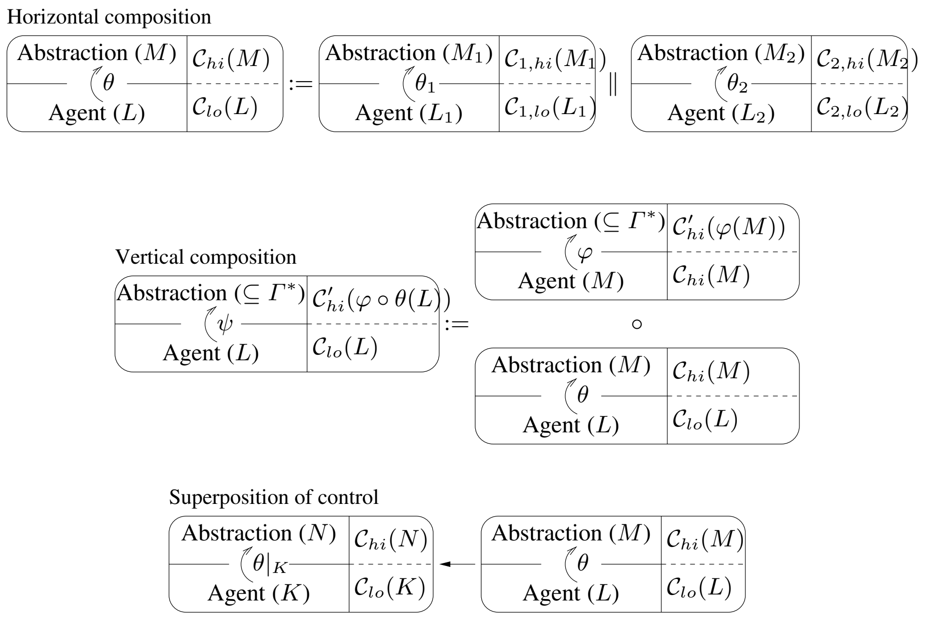


Fig. 7. Composition of components and superposition of control

Let $p_i : (\Sigma_1 \cup \Sigma_2)^* \to \Sigma_i^*$ be natural projections: $p_i(\epsilon) := \epsilon$; $p_i(\sigma) := \sigma$ if $\sigma \in \Sigma_i$; $p_i(\sigma) := \epsilon$ if $\sigma \notin \Sigma_i$; and $p_i(w\sigma) := p_i(w)p_i(\sigma)$ for $w \in (\Sigma_1 \cup \Sigma_2)^*$ and $\sigma \in \Sigma_1 \cup \Sigma_2$ $(i = 1, 2)$. Then, $\mathcal{C}_1(K_1) \parallel \mathcal{C}_2(K_2) = \mathcal{C}(K_1 \parallel K_2)$, where $K_i \subseteq \Sigma_i^*$, $\mathcal{C}_i$ is a standard, locally definable control structure on L_i $(i = 1, 2)$ and $\mathcal{C} := \mathcal{C}_1 \parallel \mathcal{C}_2$, where $p_i \circ \mathcal{C} = \mathcal{C}_i \circ p_i$. Furthermore, $p_i \circ \overline{\mathcal{C}}(L) = \overline{\mathcal{C}}_i(L_i)$ for $i = 1, 2$ (see appendix E in [30]).

Let $\theta : L \to T^*$, where $T = T_1 \cup T_2$ and $\theta := \theta_1 \parallel \theta_2$, the synchronization of θ_1 and θ_2 [25]. It can be verified that $\mathcal{C}_{hi}(M_1 \parallel M_2) \subseteq \theta(\mathcal{C}_{lo}(L_1 \parallel L_2))$ if $\mathcal{C}_{i,hi}(M_i) \subseteq \theta_i(\mathcal{C}_{i,lo}(L_i))$ $(i = 1, 2)$. Furthermore, if $\theta_i^{-1}(M_{i,m}) = L_{i,m}$ $(i = 1, 2)$, then $\theta^{-1}(M_{1,m} \parallel M_{2,m}) = L_{1,m} \parallel L_{2,m}$. In accordance with the following proposition, the observer property is also preserved.

Proposition 5.1. *(adaptation of some results in [25]) Let $\theta_i : L_i \to T_i^*$ be a causal reporter map on a closed language L_i over Σ_i $(i = 1, 2)$, where $\Sigma_1 \cap \Sigma_2 = T_1 \cap T_2 = \emptyset$. Let $L := L_1 \parallel L_2$ be the interleaving between of L_1 and L_2 (L is a closed language over $\Sigma = \Sigma_1 \cup \Sigma_2$). Let $H_i \subseteq L_i$ $(i = 1, 2)$ and $H := H_1 \parallel H_2$. Suppose that θ_i is an H_i–observer $(i = 1, 2)$. Then $\theta : L \to T^*$ is an H-observer, where $T = T_1 \cup T_2$ and $\theta := \theta_1 \parallel \theta_2$, the synchronization of θ_1 and θ_2.*

To show that the partner-freedom is preserved, it suffices to show that the control coincidence is preserved (based on an argument used in the previous section). This can be done in the particular case of the standard control technology defined by Equation 2, since if $\theta_{i,v}^{-1}(t\tau) \subseteq \Sigma_i^* \Sigma_{i,c}$ and $\omega_i^{-1}(\tau) \subseteq \Sigma_i^* \Sigma_{i,c}$, for all $t\tau \in M_i$ and $\tau \in T_{i,c}$ $(i =$

$1,2)$, then $\theta_v^{-1}(t\tau) \subseteq (\Sigma_1 \cup \Sigma_2)^*(\Sigma_{1,c} \cup \Sigma_{2,c})$ and $\omega^{-1}(\tau) \subseteq (\Sigma_1 \cup \Sigma_2)^*(\Sigma_{1,c} \cup \Sigma_{2,c})$, for all $t\tau \in M_1 \parallel M_2$ and $\tau \in T_{1,c} \cup T_{2,c}$, and $\omega : L_1 \parallel L_2 \to T_1 \cup T_2 \cup \{\epsilon\}$ defined as follows:

$$\omega(s\sigma) := \begin{cases} \omega_i(p_i(s)\sigma), \text{ if } \sigma \in \Sigma_i \text{ and } \omega_i(p_i(s)\sigma)! \ (i = 1, 2) \\ \text{undefined} \quad \text{otherwise.} \end{cases}$$

Finally, $\mathcal{C}_{lo} = \mathcal{C}_{1,lo} \parallel \mathcal{C}_{2,lo}$ and $\mathcal{C}_{hi} = \mathcal{C}_{1,hi} \parallel \mathcal{C}_{2,hi}$ as expected.

5.2 Vertical Aggregation without Additional Control

Three points must be considered in the vertical aggregation of components. First, the design of an observer for the higher component in the vertical structure is done accordingly with the procedure described in the previous section. It is denoted φ as shown in Figure 7. By using the following proposition, $\psi = \varphi \circ \theta$ is an observer for the aggregated component.

Proposition 5.2. *(lemma 5 in [31]) Let $\theta : L \to T^*$ be a causal reporter map on a closed language L over Σ and $\varphi : M \to \Gamma^*$ be a causal reporter map on $M := \theta(L)$. Let $H \subseteq L$. Suppose that θ is an H–observer and φ is a $\theta(H)$–observer. Then $\psi : L \to \Gamma^*$ is an H-observer, where $\psi := \varphi \circ \theta$.*

Second, the abstraction of the higher component is equipped with the standard control technology in accordance with Equation 2, which implies control coincidence. Consequently, weak control consistency and partner-freedom properties hold in this component. Third, the consistency of marking is simply satisfied by defining the marked languages appropriately. Therefore, from the two source components:

$\mathcal{C}_{hi}(M) \subseteq \theta(\mathcal{C}_{lo}(L))$ and $\mathcal{C}'_{hi}(\varphi(M)) \subseteq \varphi(\mathcal{C}_{hi}(M))$ (weak control consistency),

which implies $\mathcal{C}'_{hi}(\varphi(M)) \subseteq \varphi(\mathcal{C}_{hi}(M)) \subseteq \varphi \circ \theta(\mathcal{C}_{lo}(L))$;

$\theta^{-1}(\theta(L_m)) = \theta^{-1}(M_m) = L_m$ and $\varphi^{-1}(\varphi(M_m)) = M_m$ (consistency of marking); and

$\theta_v^{-1} \circ \overline{\kappa}_M \leq \kappa_L \circ \theta^{-1} \circ \overline{\kappa}_M$ and $\varphi_v^{-1} \circ \overline{\kappa}_{\varphi(M)} \leq \kappa_M \circ \varphi^{-1} \circ \overline{\kappa}_{\varphi(M)}$ (partner-freedom).

Thus, weak control consistency and consistency of marking properties are preserved in the target component:

$$\mathcal{C}'_{hi}(\psi(L)) = \mathcal{C}'_{hi}(\varphi \circ \theta(L)) \subseteq \varphi \circ \theta(\mathcal{C}_{lo}(L)) = \psi(\mathcal{C}_{lo}(L));$$
$$L_m = \theta^{-1}(\varphi^{-1}(\varphi(M_m))) = \theta^{-1}(\varphi^{-1}(\varphi(\theta(L_m)))) = \theta^{-1}(\varphi^{-1}(\psi(L_m))) = \psi^{-1}(\psi(L_m)).$$

Verifying the preservation of partner-freedom property requires more attention. Since $\kappa_M \circ \varphi^{-1} \circ \overline{\kappa}_{\varphi(M)}$ is the low-level implementation of the high-level synthesis (with respect to the higher component) of some closed controllable language of $\varphi(M)$ and partner-freedom between L and M and between M and $\varphi(M)$ holds, then

$$\theta_v^{-1}(\kappa_M \circ \varphi^{-1} \circ \overline{\kappa}_{\varphi(M)}) \leq \kappa_L \circ \theta^{-1}(\kappa_M \circ \varphi^{-1} \circ \overline{\kappa}_{\varphi(M)})$$
$$\theta_v^{-1}(\varphi_v^{-1} \circ \overline{\kappa}_{\varphi(M)}) \leq \theta_v^{-1}(\kappa_M \circ \varphi^{-1} \circ \overline{\kappa}_{\varphi(M)}) \leq \kappa_L \circ \theta^{-1}(\kappa_M \circ \varphi^{-1} \circ \overline{\kappa}_{\varphi(M)})$$

and since $\kappa_M \circ \varphi^{-1} \circ \overline{\kappa}_{\varphi(M)} \leq \varphi^{-1} \circ \overline{\kappa}_{\varphi(M)}$,

$$\theta_v^{-1}(\varphi_v^{-1} \circ \overline{\kappa}_{\varphi(M)}) \leq \kappa_L \circ \theta^{-1}(\kappa_M \circ \varphi^{-1} \circ \overline{\kappa}_{\varphi(M)}) \leq \kappa_L \circ \theta^{-1}(\varphi^{-1} \circ \overline{\kappa}_{\varphi(M)}).$$

Finally,

$$\begin{aligned} \theta_v^{-1}(\varphi_v^{-1} \circ \overline{\kappa}_{\varphi(M)}) &\leq \kappa_L \circ \theta^{-1}(\varphi^{-1} \circ \overline{\kappa}_{\varphi(M)}) \\ (\theta_v^{-1} \circ \varphi_v^{-1}) \circ \overline{\kappa}_{\varphi(M)} &\leq \kappa_L \circ (\theta^{-1} \circ \varphi^{-1}) \circ \overline{\kappa}_{\varphi(M)} \\ \psi_v^{-1} \circ \overline{\kappa}_{\varphi(\theta(L))} &\leq \kappa_L \circ \psi^{-1} \circ \overline{\kappa}_{\varphi(\theta(L))} \\ \psi_v^{-1} \circ \overline{\kappa}_{\psi(L)} &\leq \kappa_L \circ \psi^{-1} \circ \overline{\kappa}_{\psi(L)}. \end{aligned}$$

The preservation weak control consistency and partner-freedom properties should be deduced from the preservation of the control coincidence property, since

$$\theta^{-1}(\mathcal{C}_{hi}(M)) \subseteq \mathcal{C}_{lo}(L) \text{ and } \varphi^{-1}(\mathcal{C}'_{hi}(\varphi(M))) \subseteq \mathcal{C}_{hi}(M) \text{ (control coincidence)}$$

implies

$$\begin{aligned} \psi^{-1}(\mathcal{C}'_{hi}(\psi(L))) &= \psi^{-1}(\mathcal{C}'_{hi}(\varphi \circ \theta(L))) = \psi^{-1}(\mathcal{C}'_{hi}(\varphi(M))) = \theta^{-1} \circ \varphi^{-1}(\mathcal{C}'_{hi}(\varphi(M))) \\ &\subseteq \theta^{-1}(\mathcal{C}_{hi}(M)) \subseteq \mathcal{C}_{lo}(L). \end{aligned}$$

But, it has been shown that this assumption was unnecessary to establish the preservation of these two first properties.

It should be noted that no control is added during reabstraction. In fact the control structure for the agent in the higher component is the same as the one for the abstraction in the lower component.

5.3 Superposition of Control

In this case, the component has all the required properties (weak control consistency, consistency of marking, observer and partner-freedom), but a nonblocking controllable language $N = \kappa_M(E)$ that belongs to $\mathcal{C}_{hi}$ is selected to impose further control on the abstraction, where E is a high-level specification. Thus, the control associated with N in the lower level is $K = \kappa_L \circ \theta^{-1}(\overline{\kappa_M(E)})$. The control technology in the upper level is clearly the same as the one for M. Therefore, the control coincidence property is fulfilled. The only condition that remains to verify is that θ is a K-observer. Since the partner-freedom property between M and L holds, then (L, θ) is strongly observable with respect to $\mathcal{X} := \{H \in \overline{\mathcal{C}}_{lo}(L) \mid \kappa_L \circ \theta^{-1} \circ \theta(H) = H \wedge \theta(H) \in C_{hi}(M)\}$, the set of closed and controllable sublanguages that are supremal with respect to their corresponding controllable high level languages, and since $K \in \mathcal{X}$ then we have that $\theta|_K$ is an observer (see the argumentation of proposition 16 in [32]).

6 Conclusion

The approach advocated in this paper suggests to apply SCT, in particular the hierarchical control variant, to component-based software development of complex systems. If basic components are pairwise disjoint, satisfy control consistency, consistency of marking, observer and partner-freedom properties, then the theory ensures that nonblockingness is preserved during the composition of components or when further control is superposed to the original control. Furthermore, control can become explicit (and be combined), which means that the controllability of specifications must be verified or, even better, controllers can be synthesized.

Briefly, within the formal framework of the hierarchical variant of SCT, it seems to be possible to build components that foster a design process that is correct by construction (SCT synthesis procedures), is incremental in nature (localization of control), allows abstraction (well-defined interfaces) and provides support for encapsulation through abstract interfaces (abstraction rules), as in the framework of Gössler and Sifakis [11]. However, SCT being set in the context of general systems theory, it does not directly account for heterogeneity of system architectures (synchronous vs asynchronous execution paradigm, atomic vs non-atomic interactions with or without strict synchronization). In this sense the hierarchical variant of SCT is less flexible. One can contend however, that given the presence of well-defined procedures for subsystems synthesis and abstraction, this may still be worth careful examination. Nevertheless, it is endowed with a fully associative and commutative composition operator that allows for combining components incrementally into larger subsystems. As long as the procedures (and operators) for composition, synthesis and abstraction are used, one remains within the framework with all its benefits. In particular this affords for building further components by assembly of already abstract components from within the same framework, incorporating the benefits of component reuse to the very process of component building. This opens the way to pyramidal hierarchies of abstract components, potentially compounding the cost savings usually associated with reuse.

In CBSE with vertical and horizontal aggregation of components, an explicit imperative control flow mechanism is generally assumed, the service call. Thus control actions are considered to propagate downwards from the top level through the use of service calls performed on the aggregated components, hence control flows downward. In the control theory of DES there is no such mechanism assumed. In a system, events are considered to occur in no particular order or pattern, and control is exercised by allowing or disallowing the occurrence of those events which can be prevented to occur (the controllable events). Indeed, if a system is viewed as an aggregation of devices, then system's activities actually start from the devices themselves and the control flow would seem to be reversed from that of a service call based model. One may wonder if this situation could not invalidate a paradigm such as the one proposed in this paper: implementation + control + interface. It appears not to be so. In the hierarchical control of DES, control does propagate downwards from the top level under the form of constraints coming from beyond the local scope of an aggregated component (i.e., its operating environment). Those environmental constraints are added to the local control exercised by this component as further constraints for the component and eventually relayed down to the next level if required. Therefore everything happens as if there existed a control flow running form the top down and the conceptual model is preserved despite the lack of an explicit control flow mechanism. This situation means that programmers must think carefully during the design process.

The restriction concerning the alphabets of components (they must be pairwise disjoints) seems too restrictive. A richer model, which consists of a set of components, a set of channels and links between the components and channels could be considered. Even though such variants exist in the setting of SCT (e.g., [27]), substantial research effort should be devoted to combine both variants. The formalism used to model BPEL processes (automata) is another limitation, since BPEL is more expressive than automata. The use of a richer formalism like Petri nets with well defined structural properties could be used. Such restrictive assumptions are frequently made even though BPEL processes are modeled with Petri nets (e.g., [23,21]). In these pieces of research formal methods allow for verifying BPEL processes compatibility. Unfortunately, the term *controllability* used in [21] has not the same meaning as the one usually given in SCT. Furthermore, blockingness and control are not considered.

In the last decades a major preoccupation of the CBSE community was the definition of operators for combining components, based on process algebras, π-calculus or category of coalgebras, as well as algebraic laws in order to specify properties being satisfied by these operators. This aspect was not overlooked in this paper with respect to the work presented in Section 5. This was done, however, in the perspective of a control theory hoping to bring the benefits of a formal specification of the control aspect of components within the scope of the domain.

References

1. Andova, S., Groenewegen, L.P.J., de Vink, E.P.: Distributed adaptation of dining philosophers. In: Pre-proceedings of 7th International Workshop on Formal Aspects of Component Software, Guimarães Portugal, pp. 101–119 (2010)
2. Barbeau, M., Kabanza, F., St-Denis, R.: A method for the synthesis of controllers to handle safety, liveness, and real-time constraints. IEEE Transactions on Automatic Control 43, 1543–1559 (1998)
3. Bensalem, S., Bozga, M., Sifakis, J., Nguyen, T.-H.: Compositional verification for component-based systems and application. In: Cha, S., Choi, J.-Y., Kim, M., Lee, I., Viswanathan, M. (eds.) ATVA 2008. LNCS, vol. 5311, pp. 64–79. Springer, Heidelberg (2008)
4. Bherer, H., Desharnais, J., St-Denis, R.: Control of parameterized discrete event systems. Discrete Event Dynamic Systems: Theory and Applications 19, 213–265 (2009)
5. Bruneton, E., Coupaye, T., Stefani, J.B.: The Fractal component model. OW2 Consortium technical report, version 2.0-3 (2004), `http://Fractal.OW2.org`
6. Broy, M.: A theory of system interaction: components, interfaces, and services. In: Goldin, D., Smolka, S.A., Wegner, P. (eds.) Interactive Computation: the New Paradigm, pp. 41–96. Springer, Heidelberg (2006)
7. Cassandras, C.G., Lafortune, S.: Introduction to Discrete Event Systems, 2nd edn. Springer, New York (2008)
8. Chen, X., He, J., Liu, Z., Zhan, N.: A model of component-based programming. In: Arbab, F., Sirjani, M. (eds.) FSEN 2007. LNCS, vol. 4767, pp. 191–206. Springer, Heidelberg (2007)
9. Côté, D.: Conception par composantes de contrôleurs d'usines modulaires utilisant la théorie du contrôle supervisé. Ph.D. thesis, Département d'informatique, Université de Sherbrooke, submitted (2011)

10. Dormoy, J., Kouchnarenko, O., Lanoix, A.: Using temporal logic for dynamic reconfigurations of components. In: Pre-proceedings of 7th International Workshop on Formal Aspects of Component Software, Guimarães Portugal, pp. 121–138 (2010)
11. Gössler, G., Sifakis, J.: Composition for component-based modeling. Science of Computer Programming 55, 161–183 (2005)
12. Holloway, L.E., Krogh, B.H., Giua, A.: A survey of Petri net methods for controlled discrete event systems. Discrete Event Dynamic Systems: Theory and Applications 7, 151–190 (1997)
13. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation, 3rd edn. Addison-Wesley, Boston (2006)
14. Kumar, R., Garg, V.K.: Modeling and Control of Logical Discrete Event Systems. Kluwer Academic Publishers, Boston (1995)
15. Liu, Z., Morisset, C., Stolz, V.: rCOS: Theory and Tool for Component-Based Model Driven Development. In: Arbab, F., Sirjani, M. (eds.) FSEN 2009. LNCS, vol. 5961, pp. 62–80. Springer, Heidelberg (2010)
16. Lau, K.-K., Ornaghi, M., Wang, Z.: A Software Component Model and Its Preliminary Formalisation. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 1–21. Springer, Heidelberg (2006)
17. Lau, K.-K., Wang, Z.: Software component models. IEEE Transactions on Software Engineering 33, 709–724 (2007)
18. Leduc, R.J., Brandin, B.A., Lawford, M., Wonham, W.M.: Hierarchical interface-based supervisory control—part I: serial case. IEEE Transactions on Automatic Control 50, 1322–1335 (2005)
19. Leduc, R.J., Lawford, M., Wonham, W.M.: Hierarchical interface-based supervisory control—part II: parallel case. IEEE Transactions on Automatic Control 50, 1336–1348 (2005)
20. Leveson, N.G.: Engineering a safer world: system safety for the 21st century (or systems thinking applied to safety). In: Aeronautics and Astronautics and Engineering Systems Division. MIT, Cambridge (2009)
21. Lohmann, N., Massuthe, P., Stahl, C., Weinberg, D.: Analyzing interacting WS-BPEL processes using flexible model generation. Data & Knowledge Engineering 64, 38–54 (2008)
22. Ma, C., Wonham, W.M.: Nonblocking supervisory control of state tree structures. IEEE Transactions on Automatic Control 51, 782–793 (2006)
23. Martens, A., Moser, S., Gerhardt, A., Funk, K.: Analyzing compatibility of BPEL processes. In: Proceedings of Advanced International Conference on Telecommunications and International Conference on Internet and Web Applications and Services (AICT/ICIW), Guadeloupe, French Caribbean, p. 147 (2006)
24. Milner, R.: Communication and Concurrency. Prentice Hall, New York (1989)
25. Pu, K.Q.: Modeling and control of discrete-event systems with hierarchical abstraction. Master thesis, Department of Electrical and Computer Engineering, University of Toronto (2000)
26. Ramadge, P.J., Wonham, W.M.: Control of discrete event systems. Proceedings of the IEEE 77, 81–98 (1989)
27. Santos, E.A.P., De Negri, V.J., Cury, J.E.R.: A computational model for supporting conceptual design of automatic systems. In: Proceedings of 13th International Conference on Engineering Design, Glasgow UK, pp. 517–524 (2001)
28. St-Denis, R.: Designing reactive systems: integration of abstraction techniques into a synthesis procedure. The Journal of Systems and Software 60, 103–112 (2002)
29. Szyperski, C., Gruntz, D., Murer, S.: Component Software — Beyond Object-Oriented Programming, 2nd edn. ACM Press and Addison-Wesley, New York (2002)

30. Wong, K.C.: Discrete-event control architecture: an algebraic approach. Ph.D. thesis, Department of Electrical Engineering, University of Toronto (1994)
31. Wong, K.C., Thistle, J.G., Malhamé, R.P., Hoang, H.-H.: Supervisory control of distributed systems: conflict resolution. Discrete Event Dynamic Systems: Theory and Applications 10, 131–186 (2000)
32. Wong, K.C., Wonham, W.M.: Hierarchical control of discrete-event systems. Discrete Event Dynamic Systems: Theory and Applications 6, 241–273 (1996)
33. Wong, K.C., Wonham, W.M.: On the computation of observers in discrete-event systems. Discrete Event Dynamic Systems: Theory and Applications 14, 55–107 (2004)

Aspect Weaving in UML Activity Diagrams: A Semantic and Algorithmic Framework

Djedjiga Mouheb, Dima Alhadidi, Mariam Nouh, Mourad Debbabi, Lingyu Wang[1], and Makan Pourzandi[2]

[1] Computer Security Laboratory
Concordia Institute for Information Systems Engineering
Concordia University
Montreal, Canada
{d_mouheb,dm_alhad,m_nouh,debbabi,wang}@encs.concordia.ca
[2] Ericsson Canada Inc.
Montreal, Canada
makan.pourzandi@ericsson.com

Abstract. Aspect-Oriented Modeling (AOM) is an emerging solution for handling crosscutting concerns at the software modeling level in order to reduce the complexity of software models and application code. Most existing work on weaving aspects into UML design models is presented from a practical perspective and lacks formal syntax and semantics. In this paper, we propose formal specifications for aspect weaving into UML activity diagrams and the implementation strategies of the proposed weaving semantics. To this end, we define syntax for activity diagrams and UML aspects. We also show the correctness and the completeness of the matching and the weaving processes in terms of the semantics and the algorithms provided in this paper. Finally, we demonstrate the viability and the relevance of our propositions using a case study.

Keywords: Aspect-Oriented Modeling (AOM), UML Activity Diagram, Weaving, Operational Semantics.

1 Introduction

Dealing with crosscutting concerns, such as logging and synchronization, is a major challenge in the development of software systems. In this respect, Aspect-Oriented Programming (AOP) is an appealing approach that allows the separation of crosscutting concerns from the software core functionality [15]. Due to the increasing interest, AOP has recently stretched over earlier stages of the software development life-cycle. Aspect-Oriented Modeling (AOM) [9] applies aspect-oriented techniques to software models with the aim of modularizing crosscutting concerns. Indeed, handling those concerns at the modeling level would significantly help in alleviating the complexity of software models and application code. Additionally, it reduces development costs and maintenance time.

Most of the contributions [10, 12, 21, 13, 11, 20, 17] that have explored aspect weaving into UML design models are presented from a practical perspective. The research proposals that have handled the theoretical foundations in this area are still behind practical implementation ones. Accordingly, there is a desideratum to provide such foundations that are important to offer complete and rigorous definitions for better understandability, to establish theoretical properties, and to facilitate mathematical reasoning. In this paper, we propose formal specifications for aspect weaving in UML activity diagrams and the implementation strategies of the proposed weaving semantics. We focus on activity diagram [8] typically used to model business processes and operational workflows of systems.

The concepts of our defined AOM approach are similar to the ones of AOP, namely, adaptations, join points, and pointcuts. An adaptation specifies the modification to be performed on the base model. A join point is a location in the base model where an adaptation should be applied. A pointcut is an expression that designates a set of join points. Our definition of adaptations and pointcuts is completely based on UML constructs. We support two types of adaptations: (1) add adaptations, which add new behaviors to activity diagrams before, after, or around specific join points, and (2) remove adaptations, which delete existing behaviors from activity diagrams. The novelty of our join point model is that it considers not only executable nodes, i.e., action nodes, but also various control nodes, i.e., initial, final, flow final, fork, join, decision and merge nodes. Capturing such control nodes allows modeling crosscutting concerns needed with alternatives, loops, exceptions, and multithreaded applications. On the other hand, the main advantage of the design and the implementation over existing practical efforts [10, 12, 21, 13, 11, 20, 17] is the use of the Object Constraint Language (OCL) [6] for join point matching due to its expressiveness and conformance to UML. This language is a declarative one for describing rules that apply to UML models. Furthermore, we choose Query/View/Transformation (QVT) language [5] for weaving since it is an OMG standard compatible with UML and supports a large set of modifications on UML models. Employing OCL and QVT, which are standard languages, enables the portability and the expressiveness of the proposed weaver.

The main contributions of this paper are three fold. First, we formalize semantics and algorithms for matching and weaving in UML activity diagrams. For this reason, syntax for activity diagrams together with syntax for UML aspects are defined. Second, We show the correctness and the completeness of the matching and weaving processes with respect to the semantics and algorithms. Third, we present implementation strategies of the weaving capabilities according to the proposed semantics. To explore the viability and the relevance of the defined approach, a case study is developed.

The remainder of this paper is structured as follows. Section 2 presents the syntax of UML activity diagrams and aspects. In Section 3, we define formal semantics for aspect matching and weaving. A correctness and a completeness analysis is presented in Section 4. Afterwards, we explain the implementation strategies of UML weaving capabilities in Section 5. A case study is conducted in Section 6. We discuss the related work in Section 7. Concluding remarks as well as future work are represented in Section 8.

2 Syntax

This section presents the syntax of UML activity diagrams and aspects. The proposed syntax covers all the constructs that are required for the weaving semantics. This semantics describes how to inject adaptations at specific locations in the activity diagrams. First, we introduce the following notations that will be used throughout this paper.

Notation

- The algorithms and notations are written with respect to the OCaml notations [1].
- Given a record space $D = \langle f_1 : D_1, f_2 : D_2, \ldots, f_n : D_n \rangle$ and an element e of type D, the access to the field f_i of an element e is written as $e.f_i$.
- Given a type τ, we write τ`-set` to denote the type of sets having elements of type τ.
- Given a type τ, we write τ`-uset` to denote the type of sets having a unary element of type τ.
- Given a type τ, we write τ`-list` to denote the type of lists having elements of type τ.
- The type *Identifier* classifies identifiers.

2.1 Activity Diagram Syntax

An activity diagram, as shown in Fig. 1, consists of a set of nodes and a set of edges. An edge is a directed connection between two nodes represented by source and target. In addition, an edge may have a guard condition specifying if the edge can be traversed. A node can be either an executable node (e.g., action, structured activity) or a control node (e.g., initial, final). We consider the following nodes:

- Initial: represents an initial node at which the activity starts executing. It has one outgoing edge and no incoming edges.
- Final: represents a final node that can be either: (1) an activity final, at which the activity execution terminates, or (2) a flow final, at which a flow terminates. It has one incoming edge and no outgoing edges.
- Action: represents an action node. It has one incoming and one outgoing edge.
- Fork/Decision: represent fork and decision nodes. Both have one incoming edge and multiple outgoing edges.
- Join/Merge: represent join and merge nodes. Both have one outgoing edge and multiple incoming edges.
- StructuredActivity: represents a structured activity node, which may have in turn its own nodes and edges. It has one incoming and one outgoing edge.

$$\begin{array}{lll lll}
\mathit{Activity} & ::= & \langle \mathit{name}{:}\mathit{Identifier}, & \mathit{ForkDecision} & ::= & \langle \mathit{type}{:}\texttt{fork} \mid \texttt{decision}, \\
 & & \ \mathit{nodes}{:}\mathit{Node}\texttt{-set}, & & & \ \mathit{name}{:}\mathit{Identifier}, \\
 & & \ \mathit{edges}{:}\mathit{Edge}\texttt{-set}\rangle & & & \ \mathit{incoming}{:}\mathit{Edge}\texttt{-uset}, \\
\mathit{Node} & ::= & \mathit{Initial} \mid \mathit{Final} \mid & & & \ \mathit{outgoing}{:}\mathit{Edge}\texttt{-set}\rangle \\
 & & \mathit{Action} \mid & \mathit{JoinMerge} & ::= & \langle \mathit{type}{:}\texttt{join} \mid \texttt{merge}, \\
 & & \mathit{ForkDecision} \mid & & & \ \mathit{name}{:}\mathit{Identifier}, \\
 & & \mathit{JoinMerge} \mid & & & \ \mathit{incoming}{:}\mathit{Edge}\texttt{-set}, \\
 & & \mathit{Structured} & & & \ \mathit{outgoing}{:}\mathit{Edge}\texttt{-uset}\rangle \\
\mathit{Initial} & ::= & \langle \mathit{type}{:}\texttt{initial}, & \mathit{Structured} & ::= & \langle \mathit{type}{:}\texttt{structuredactivity}, \\
 & & \ \mathit{name}{:}\mathit{Identifier}, & & & \ \mathit{name}{:}\mathit{Identifier}, \\
 & & \ \mathit{outgoing}{:}\mathit{Edge}\texttt{-uset}\rangle & & & \ \mathit{incoming}{:}\mathit{Edge}\texttt{-uset}, \\
\mathit{Final} & ::= & \langle \mathit{type}{:}\texttt{final} \mid \texttt{flowfinal}, & & & \ \mathit{outgoing}{:}\mathit{Edge}\texttt{-uset}, \\
 & & \ \mathit{name}{:}\mathit{Identifier}, & & & \ \mathit{nodes}{:}\mathit{Node}\texttt{-set}, \\
 & & \ \mathit{incoming}{:}\mathit{Edge}\texttt{-uset}\rangle & & & \ \mathit{edges}{:}\mathit{Edge}\texttt{-set}\rangle \\
\mathit{Action} & ::= & \langle \mathit{type}{:}\texttt{action}, & \mathit{Edge} & ::= & \langle \mathit{name}{:}\mathit{Identifier}, \\
 & & \ \mathit{name}{:}\mathit{Identifier}, & & & \ \mathit{source}{:}\mathit{Node}, \\
 & & \ \mathit{incoming}{:}\mathit{Edge}\texttt{-uset}, & & & \ \mathit{target}{:}\mathit{Node}, \\
 & & \ \mathit{outgoing}{:}\mathit{Edge}\texttt{-uset}\rangle & & & \ \mathit{guard}{:}\texttt{true} \mid \texttt{false}\rangle
\end{array}$$

Fig. 1. Activity Diagram Syntax

2.2 Aspect Syntax

An aspect, as depicted in Fig. 2, includes a list of adaptations. An adaptation can be of two kinds:

- Add adaptation: includes the following:
  - The activity element to be injected at specific locations picked out by pointcuts. It can be either a basic element (action) or a composed element (structured activity).
  - The insertion point that specifies where the activity element should be injected according to a specific location. It can have the following three

$$\begin{array}{lllr}
\mathit{Aspect} & ::= & \mathit{Adaptation}\texttt{-list} & \textsf{(Aspect)} \\
\mathit{Adaptation} & ::= & \langle \mathit{kind}{:}\ \texttt{add}, & \textsf{(Adaptations)} \\
 & & \ \mathit{elem}{:}\ \mathit{Action} \mid \mathit{Structured}, & \\
 & & \ \mathit{pos}{:}\ \texttt{before} \mid \texttt{after} \mid \texttt{around}, & \\
 & & \ \mathit{pcd}{:}\ \mathit{Pcd}\rangle & \\
 & \mid & \langle \mathit{kind}{:}\ \texttt{remove}, & \\
 & & \ \mathit{pcd}{:}\ \mathit{Pcd}\rangle & \\
\mathit{Pcd} & ::= & \texttt{true} & \textsf{(Pointcuts)} \\
 & \mid & \neg p & \\
 & \mid & p \wedge p & \\
 & \mid & \langle \mathit{kind}{:}\ \texttt{initial} \mid \texttt{final} \mid \texttt{flowfinal} \mid \texttt{action} \mid \texttt{fork} & \\
 & & \quad \mid \texttt{join} \mid \texttt{decision} \mid \texttt{merge} \mid \texttt{inside\_activity}, & \\
 & & \ \mathit{name}{:}\ \mathit{Identifier}\rangle &
\end{array}$$

Fig. 2. Aspect Syntax

values: before, after, and around. A before- (resp. after-) position means that the new element should be added before (resp. after) the identified location, while an around-position means that the existing element at the identified location should be replaced with a new one.

– Remove adaptation: includes a pointcut that picks out the elements that should be removed from the activity diagram.

A pointcut specifies a set of join points in the activity diagram where the aspect adaptations should be applied. We consider the following kinds of basic pointcuts: `initial`, `final`, `flowfinal`, `action`, `fork`, `join`, `decision`, `merge`, and `inside_activity`. These basic pointcuts can be combined with logical operators to produce more complex ones.

3 Matching and Weaving Semantics

In this section, we present the matching and the weaving semantics in activity diagrams. The matching semantics describes how to identify the join points targeted by the activity adaptations. The weaving semantics describes how to apply the activity adaptations at the identified join points.

3.1 Matching Semantics

We define the judgment $\mathcal{A}, n \vdash_{match} pcd$, , which is used in the matching semantic rules presented in Fig. 3, to describe that a node n belonging to the activity $\mathcal{A}$ matches the pointcut *pcd*. A node n can be an initial node i, an activity final node *af*, a flow final node *ff*, an action node a, a fork node f, a join node j, a decision node d, a merge node m, or either of these nodes *sn*. In the following, we explain the matching semantic rules:

$$\frac{pcd.kind = \texttt{initial} \qquad pcd.name = i.name}{\mathcal{A}, i \vdash_{match} pcd} \qquad \frac{pcd.kind = \texttt{final} \qquad pcd.name = af.name}{\mathcal{A}, af \vdash_{match} pcd}$$

$$\frac{pcd.kind = \texttt{flowfinal} \qquad pcd.name = ff.name}{\mathcal{A}, ff \vdash_{match} pcd} \qquad \frac{pcd.kind = \texttt{action} \qquad pcd.name = a.name}{\mathcal{A}, a \vdash_{match} pcd}$$

$$\frac{pcd.kind = \texttt{fork} \qquad pcd.name = f.name}{\mathcal{A}, f \vdash_{match} pcd} \qquad \frac{pcd.kind = \texttt{join} \qquad pcd.name = j.name}{\mathcal{A}, j \vdash_{match} pcd}$$

$$\frac{pcd.kind = \texttt{decision} \qquad pcd.name = d.name}{\mathcal{A}, d \vdash_{match} pcd} \qquad \frac{pcd.kind = \texttt{merge} \qquad pcd.name = m.name}{\mathcal{A}, m \vdash_{match} pcd}$$

$$\frac{pcd.kind = \texttt{inside\_activity} \qquad pcd.name = \mathcal{A}.name}{\mathcal{A}, sn \vdash_{match} pcd}$$

$$\frac{\mathcal{A}, n \vdash_{match} pcd_1 \qquad \mathcal{A}, n \vdash_{match} pcd_2}{\mathcal{A}, n \vdash_{match} pcd_1 \ \wedge \ pcd_2} \qquad \frac{\mathcal{A}, n \vdash_{match} pcd_1}{\mathcal{A}, n \vdash_{match} pcd_1 \ \vee \ pcd_2}$$

$$\frac{\mathcal{A}, n \vdash_{match} pcd_2}{\mathcal{A}, n \vdash_{match} pcd_1 \ \vee \ pcd_2} \qquad \frac{\mathcal{A}, n \nvdash_{match} pcd}{\mathcal{A}, n \vdash_{match} \ \neg pcd}$$

Fig. 3. Matching Semantics

Initial Describes the case where the current node is an initial node, the current pointcut is an initial one, and the pointcut name equals the node name. In such a case, the initial node matches the pointcut.

Final Describes the case where the current node is an activity final node, the current pointcut is a final one, and the pointcut name equals the node name. In such a case, the activity final node matches the pointcut.

FlowFinal Describes the case where the current node is a flow final node, the current pointcut is a flow final one, and the pointcut name equals the node name. In such a case, the flow final node matches the pointcut.

Action Describes the case where the current node is an action node, the current pointcut is an action one, and the pointcut name equals the node name. In such a case, the action node matches the pointcut.

Fork Describes the case where the current node is a fork node, the current pointcut is a fork one, and the pointcut name equals the node name. In such a case, the fork node matches the pointcut.

Join Describes the case where the current node is a join node, the current pointcut is a join one, and the pointcut name equals the node name. In such a case, the join node matches the pointcut.

Decision Describes the case where the current node is a decision node, the current pointcut is a decision one, and the pointcut name equals the node name. In such a case, the decision node matches the pointcut.

Merge Describes the case where the current node is a merge node, the current pointcut is a merge one, and the pointcut name equals the node name. In such a case, the merge node matches the pointcut.

InsideActivity Describes the case where the current node is a *sn* node, i.e., initial, final, flow final, action, fork, join, decision, or merge node, the current pointcut is an inside_activity one, and the pointcut name equals the name of the activity containing the node. In such a case, the *sn* node matches the pointcut.

And, Or_1, Or_2, **and** Not Describe the cases where pointcuts are combined using logical operators to produce more complex ones.

3.2 Weaving Semantics

The weaving semantics presented in Fig. 4 is represented by the weaving configuration $\langle \mathit{Activity}, \mathit{Aspect}, \mathit{Node}, \mathit{State}\rangle$. The state *State* is a flag that represents the stage of the adaptation weaving process, which is either `weaving` or `end`. The flag is equal to `weaving` when adaptations still have to be woven whereas it becomes `end` when the weaving is completed. Hence, the transformation $\langle \mathcal{A}, s, n, \mathtt{weaving}\rangle \hookrightarrow \langle \mathcal{A}', [\,], n', \mathtt{end}\rangle$ means that the activity diagram $\mathcal{A}'$ is the result of weaving all the applicable adaptations in the adaptation list s into the node n. Before presenting the weaving rules, we define a function builtEdge that takes two nodes as inputs and returns an edge between these two nodes:

$\mathsf{builtEdge} : \mathit{Node} \times \mathit{Node} \rightarrow \mathit{Edge}$
$\mathsf{builtEdge}(s, t) = e$ where $(e.\mathit{source} = s) \wedge (e.\mathit{target} = t)$

$$\frac{\begin{array}{c} s = ad :: s' \qquad ad.kind = \texttt{add} \qquad ad.pos = \texttt{before} \qquad n.type \neq \texttt{initial} \\ \mathcal{A}, n \vdash_{match} ad.pcd \qquad es = n.incoming \qquad e \in es \qquad e.target = ad.elem \\ e' = builtEdge(ad.elem, n) \qquad ad' = \{ad \; with \; elem.incoming = e, elem.outgoing = e'\} \\ n' = \{n \; with \; incoming = (es \backslash \{e\}) \cup \{e'\}\} \qquad no = \mathcal{A}.nodes \qquad ed = \mathcal{A}.edges \\ \mathcal{A}' = \{\mathcal{A} \; with \; nodes = (no \backslash \{n\}) \cup \{n', ad'.elem\}, edges = ed \cup \{e'\}\} \end{array}}{\langle \mathcal{A}, s, n, \texttt{weaving} \rangle \hookrightarrow \langle \mathcal{A}', s', n', \texttt{weaving} \rangle} \quad (\mathsf{Before})$$

$$\frac{\begin{array}{c} s = ad :: s' \qquad ad.kind = \texttt{add} \qquad ad.pos = \texttt{after} \qquad n.type \neq \texttt{final} \qquad n.type \neq \texttt{flowfinal} \\ \mathcal{A}, n \vdash_{match} ad.pcd \qquad os = n.outgoing \qquad e \in os \qquad next = e.target \\ e' = builtEdge(ad.elem, next) \qquad e.target = ad.elem \\ ad' = \{ad \; with \; elem.incoming = e, elem.outgoing = e'\} \qquad es = next.incoming \\ next.incoming = (es \backslash \{e\}) \cup \{e'\} \qquad no = \mathcal{A}.nodes \qquad ed = \mathcal{A}.edges \\ \mathcal{A}' = \{\mathcal{A} \; with \; nodes = no \cup \{ad'.elem\}, edges = ed \cup \{e'\}\} \end{array}}{\langle \mathcal{A}, s, n, \texttt{weaving} \rangle \hookrightarrow \langle \mathcal{A}', s', n, \texttt{weaving} \rangle} \quad (\mathsf{After})$$

$$\frac{\begin{array}{c} s = ad :: s' \qquad ad.kind = \texttt{add} \qquad ad.pos = \texttt{around} \qquad n.type = \texttt{action} \\ \mathcal{A}, n \vdash_{match} ad.pcd \qquad e \in n.incoming \qquad e' \in n.outgoing \qquad e.target = ad.elem \\ e'.source = ad.elem \qquad no = \mathcal{A}.nodes \qquad ad' = \{ad \; with \; elem.incoming = e, \\ elem.outgoing = e'\} \qquad \mathcal{A}' = \{\mathcal{A} \; with \; nodes = (no \backslash \{n\}) \cup \{ad'.elem\}\} \end{array}}{\langle \mathcal{A}, s, n, \texttt{weaving} \rangle \hookrightarrow \langle \mathcal{A}', s', ad'.elem, \texttt{weaving} \rangle} \quad (\mathsf{Around})$$

$$\frac{\begin{array}{c} s = ad :: s' \qquad ad.kind = \texttt{remove} \qquad n.type = \texttt{action} \qquad \mathcal{A}, n \vdash_{match} ad.pcd \\ e \in n.incoming \qquad e' \in n.outgoing \qquad next = e'.target \qquad e.target = next \\ no = \mathcal{A}.nodes \qquad ed = \mathcal{A}.edges \qquad es = next.incoming \\ next.incoming = (es \backslash \{e'\}) \cup \{e\} \qquad \mathcal{A}' = \{\mathcal{A} \; with \; nodes = no \backslash \{n\}, edges = ed \backslash \{e'\}\} \end{array}}{\langle \mathcal{A}, s, n, \texttt{weaving} \rangle \hookrightarrow \langle \mathcal{A}', s', next, \texttt{weaving} \rangle} \quad (\mathsf{Remove})$$

$$\frac{s = ad :: s' \qquad \mathcal{A}, n \vdash_{match} \neg \; ad.pcd}{\langle \mathcal{A}, s, n, \texttt{weaving} \rangle \hookrightarrow \langle \mathcal{A}, s', n, \texttt{weaving} \rangle} \quad (\mathsf{NoMatch})$$

$$\frac{s = [\;]}{\langle \mathcal{A}, s, n, \texttt{weaving} \rangle \hookrightarrow \langle \mathcal{A}, [\;], n, \texttt{end} \rangle} \quad (\mathsf{End})$$

Fig. 4. Weaving Semantics

Before Describes the case where an add and a before adaptation matches a specific node. This adaptation can be applied before this matched node unless it is an initial node since this node starts the activity execution. The activity element of the adaptation is inserted before the matched node.

After Describes the case where an add and an after adaptation matches a specific node. This adaptation can be applied after this matched node unless it is a final node or a flow final node since those nodes terminate the activity execution. The activity element of the adaptation is inserted after the matched node.

Around Describes the case where an add and an around adaptation matches a specific node. This adaptation can be applied just around matched action nodes. The activity element of the adaptation supersedes the matched node.

Remove Describes the case where a remove adaptation matches a specific node. This adaptation can be applied just on matched action nodes. The matched node is deleted from the activity diagram.

NoMatch Describes the case where the current adaptation pointcut does not match a node n. In this case, the activity diagram remains the same and the weaving process continues with the rest of the adaptations.

End describes the case where there are no more adaptations to apply on the activity diagram. In this case, the activity diagram remains the same and the weaving process terminates.

4 Completeness and Correctness of the Weaving

In this section, we address the correctness and the completeness of the weaving in UML activity diagrams. We first present algorithms that implement the matching and the weaving semantics reported in the rules in Fig. 3 and Fig. 4 respectively. Since the semantics in general is syntax-directed, it is a well-known fact that it can be turned into algorithms. Accordingly, the proofs of the correctness and the completeness of the matching and weaving processes according to the provided semantics and algorithms are straightforward and we choose to omit them for space limitation.

The matching algorithm $\mathcal{M}$ is presented in Fig. 5. It takes three arguments: an activity diagram $\mathcal{A}$, a node n, and a pointcut *pcd*. It returns true if the node n matches the pointcut *pcd* and it returns false otherwise. The weaving algorithm $\mathcal{W}$ is presented in Fig. 6. It takes three arguments: an activity diagram $\mathcal{A}$, an adaptation list s, and a node n. The outcome of the weaving algorithm is an activity diagram $\mathcal{A}'$ that represents the woven diagram.

```
$\mathcal{M}(\mathcal{A}, n, pcd)$ = case $pcd.kind$ of

 inside_activity                      ⇒ if $n.type \in$ {initial,final,flowfinal,
                                          action,fork,join,decision,merge} then
                                          $pcd.name = \mathcal{A}.name$
 initial| final| flowfinal|
  action| fork| join| decision| merge ⇒ if $n.type = pcd.kind$ then
                                          $n.name = pcd.name$
```

Fig. 5. Matching Algorithm

Lemma 1. *(Soundness of $\mathcal{M}$). Given an activity diagram $\mathcal{A}$, an activity node n, and a pointcut pcd. If $\mathcal{M}(\mathcal{A}, n, pcd)$ then $\mathcal{A}, n \vdash_{match} pcd$.*

Proof. The proof of Lemma 1 is straightforward since the algorithm $\mathcal{M}$ results from the rules presented in Fig. 3.

Lemma 2. *(Completeness of $\mathcal{M}$). Given an activity diagram $\mathcal{A}$, an activity node n, and a pointcut pcd. If $\mathcal{A}, n \vdash_{match} pcd$ then $\mathcal{M}(\mathcal{A}, n, pcd)$.*

Proof. The proof of Lemma 2 is straightforward since the algorithm $\mathcal{M}$ results from the rules presented in Fig. 3.

Theorem 1. *(Soundness of $\mathcal{W}$). Given an activity diagram $\mathcal{A}$, an adaptation list s, and a node n. If $\mathcal{W}(\mathcal{A}, s, n){=}\mathcal{A}''$ then $\langle \mathcal{A}, s, n, \texttt{weaving}\rangle \hookrightarrow \langle \mathcal{A}'', [\,], n'', \texttt{end}\rangle$.*

Proof. The proof of Theorem 1 is straightforward since the algorithm $\mathcal{W}$ results from the rules presented in Fig. 4.

Theorem 2. *(Completeness of $\mathcal{W}$). Given an activity diagram $\mathcal{A}$, an adaptation list s, and a node n.*
If $\langle \mathcal{A}, s, n, \texttt{weaving}\rangle \hookrightarrow \langle \mathcal{A}'', [\,], n'', \texttt{end}\rangle$ then $\mathcal{W}(\mathcal{A}, s, n){=}\mathcal{A}''$.

Proof. The proof of Theorem 2 is straightforward since the algorithm $\mathcal{W}$ results from the rules presented in Fig. 4.

5 Design and Implementation

We design and implement the weaving features that are inspired from the defined semantics as a plug-in to the IBM Rational Software Modeler tool (RSM v7.5.2) [14]. The steps and the technologies that are followed to implement the weaving capabilities are presented in Fig. 7. The implemented weaving process is organized into three main steps: (1) aspects specialization, (2) join point matching, and (3) weaving. In the following, we explain each of these steps.

5.1 Aspect Specialization

For the purpose of reuse, aspects are designed as generic templates representing crosscutting concerns independently of the application specificities. Since generic pointcuts have no concrete specification, an aspect needs to be specialized to a specific application before it can be woven into base models. To this end, we provide a weaving interface that exposes the generic pointcuts to the developer. From this weaving interface and based on his/her understanding of the application, the developer has the possibility of mapping each generic element of the aspect to its corresponding element(s) in the base model. After mapping all the generic elements, the application-dependent aspect is automatically generated by the defined framework.

5.2 Matching

During the matching, the join points where aspect adaptations should be applied are automatically selected from the base model. The targeted join points are actions and control nodes. In order to identify and match join points, we translate the pointcuts of the application-dependent aspect into a language that can easily navigate the activity diagram and query its elements. We choose Object Constraint Language (OCL) [6] due to its expressiveness and conformance to UML.

```
$\mathcal{W}(\mathcal{A}, s, n)$ = case $s$ of
 $ad :: s'$ ⇒ if $\mathcal{M}(\mathcal{A}, n, ad.pcd)$ then
             case $ad.kind$ of
              add ⇒ case $ad.pos$ of
                 before ⇒ if $n.type \neq$ initial && $e \in n.incoming$ then
                            let $es = n.incoming$
                                $e.target = ad.elem$
                                $e' = builtEdge(ad.elem, n)$
                                $ad' = \{ad\ with\ elem.incoming = e, elem.outgoing = e'\}$
                                $n' = \{n\ with\ incoming = (es \backslash \{e\}) \cup \{e'\}\}$
                                $no = \mathcal{A}.nodes$
                                $ed = \mathcal{A}.edges$
                                $\mathcal{A}' = \{\mathcal{A}\ with\ nodes = (no \backslash \{n\}) \cup \{n', ad'.elem\}, edges = ed \cup \{e'\}\}$
                            in $\mathcal{W}(\mathcal{A}', s', n')$
                 after  ⇒ if $n.type \neq$ final && $n.type \neq$ flowfinal && $e \in n.outgoing$ then
                            let $os = n.outgoing$
                                $next = e.target$
                                $e' = builtEdge(ad.elem, next)$
                                $e.target = ad.elem$
                                $ad' = \{ad\ with\ elem.incoming = e, elem.outgoing = e'\}$
                                $es = next.incoming$
                                $next.incoming = (es \backslash \{e\}) \cup \{e'\}$
                                $no = \mathcal{A}.nodes$
                                $ed = \mathcal{A}.edges$
                                $\mathcal{A}' = \{\mathcal{A}\ with\ nodes = no \cup \{ad'.elem\}, edges = ed \cup \{e'\}\}$
                            in $\mathcal{W}(\mathcal{A}', s', n)$
                 around ⇒ if $n.type =$ action && $e \in n.incoming$ && $e' \in n.outgoing$ then
                            let $e.target = ad.elem$
                                $e'.source = ad.elem$
                                $no = \mathcal{A}.nodes$
                                $ad' = \{ad\ with\ elem.incoming = e, elem.outgoing = e'\}$
                                $\mathcal{A}' = \{\mathcal{A}\ with\ nodes = (no \backslash \{n\}) \cup \{ad'.elem\}\}$
                            in $\mathcal{W}(\mathcal{A}', s', ad'.elem)$
              remove ⇒ if $n.type =$ action && $e \in n.incoming$ && $e' \in n.outgoing$ then
                         let $next = e'.target$
                             $e.target = next$
                             $no = \mathcal{A}.nodes$
                             $ed = \mathcal{A}.edges$
                             $es = next.incoming$
                             $next.incoming = (es \backslash \{e'\}) \cup \{e\}$
                             $\mathcal{A}' = \{\mathcal{A}\ with\ nodes = no \backslash \{n\}, edges = ed \backslash \{e'\}\}$
                         in $\mathcal{W}(\mathcal{A}', s', next)$
            else $\mathcal{W}(\mathcal{A}, s', n)$
 [ ]        ⇒ $\mathcal{A}$
```

Fig. 6. Weaving Algorithm

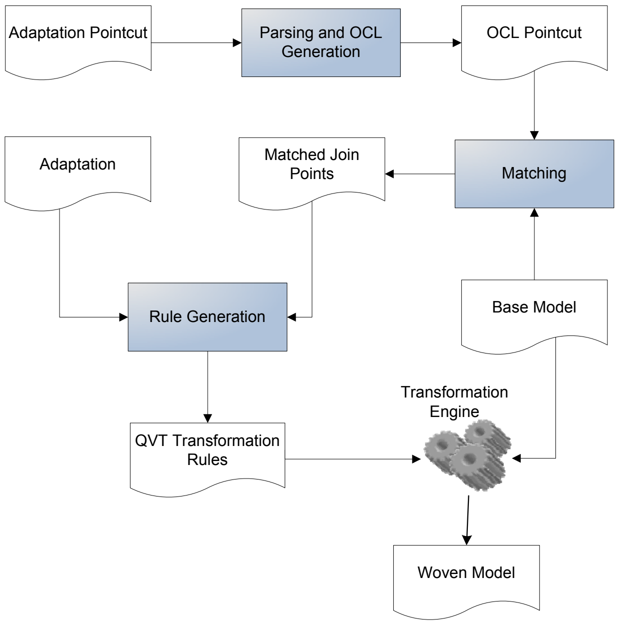


Fig. 7. Overview of the Weaving Process

In this context, we use CUP parser generator for Java [2] for translating pointcuts into OCL. This tool takes as inputs: (1) the grammar of pointcuts along with the actions required to translate each basic pointcut into its corresponding OCL expression, and (2) a scanner used to break the pointcut expression into meaningful tokens. It provides as output a parser that is capable of parsing and translating any pointcut expression into its equivalent OCL one.

Once the OCL expressions are generated, a join point matching module is called to evaluate them on the activity diagram elements. This module takes an OCL expression along with an activity element as inputs and generates a query that will evaluate the given expression on that element according to the matching semantics presented in Fig. 3. The result is a boolean value representing whether the element is a matched join point or not.

5.3 Weaving

During the weaving, aspect adaptations are automatically applied on the base model at the matched join points. We implement aspect weaving into activity diagrams as a model-to-model transformation. The latter is the process of generating target model(s) from source model(s). In our framework, the source models of the transformation are the base model and the specialized aspect model, and the target model is the woven model. We choose Query/View/Transformation (QVT) language [5] as the transformation language since it is an OMG standard compatible with UML and supports a large set of modifications on UML models. We implement the weaving capabilities using the Eclipse implementation of the QVT standard: QVT Operational (QVTO) [3] that we installed on top of Rational Software Modeler.

Typically, a model-to-model transformation consists of a set of transformation rules, also called mapping rules, which are used to describe how each element in the source model is transformed in the target model. The adopted implementation methodology consists of first translating each adaptation into its equivalent QVT mapping rule. This is done by identifying the type of adaptation, whether it is an add adaptation or a remove adaptation, and the position of the adaptation, whether it is before, after, or around. This implementation depends on the defined weaving semantics presented in Fig. 4. Once the appropriate QVT mapping rules are identified and the join points where to perform the weaving are determined, the QVTO transformation engine executes the appropriate mapping rules on the identified join points and produces as a result the woven activity diagram.

6 Case Study: Adding Authorization to SIP-Communicator

SIP-Communicator [7] is an open source software that provides internet-based audio/video telephony and instant messaging services. It is composed of more than 1400 Java classes and 150K lines of code based on the version 1.0. We present next how to add an authorization mechanism into the design models of SIP-Communicator to allow communication between only authorized clients using our framework.

The activity diagram presented in Fig. 8 depicts the specification of sending an instant message using SIP protocol. The action named `SendRequest` that invokes the method `sendRequest()` is responsible for sending a request message. This method is being called in 32 different places inside functions implementing the operations of SIP communicator, i.e., instant messaging, telephony, presence, notification, etc. The activity diagram, presented in Fig. 8, is an example showing just one occurrence of this method call. An authorization mechanism is required before any execution of the action `SendRequest`. For this purpose, we catch all the `SendRequest` actions in the design models and automatically inject the authorization mechanism into the appropriate locations using our defined framework.

The authorization aspect presented in Fig. 9 is designated using our defined UML extension for Aspect-Oriented Modeling [18]. It is based on Role-Based Access Control (RBAC) security model [19], which is an approach to restricting system access to authorized users. The authorization aspect specifies the addition of an access control behavior that checks client permissions based on the information contained in a message request. This is accomplished by defining the adaptation `AddCheckPermission` that specifies to inject the authorization behavior

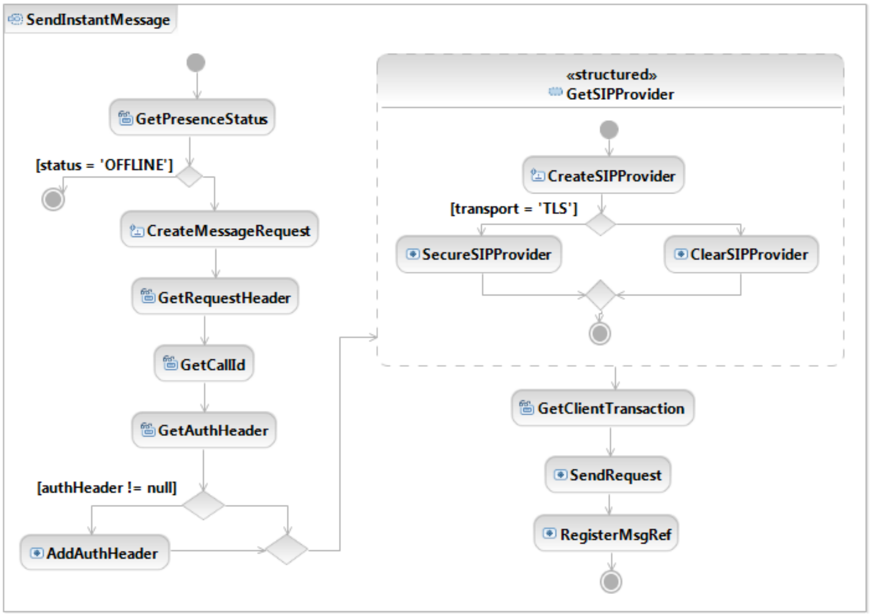


Fig. 8. Activity Diagram for Sending an Instant Message - Base Model

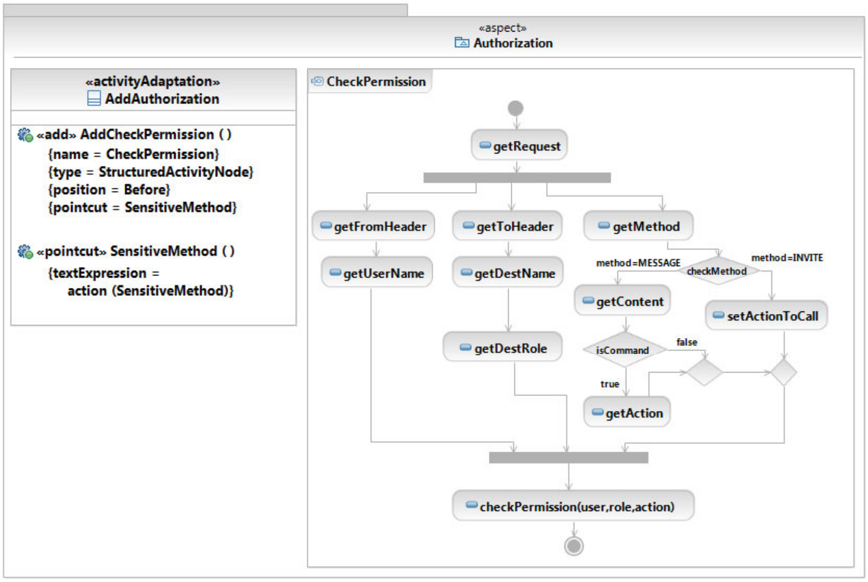


Fig. 9. Authorization Aspect

as a structured activity node before any call to a sensitive method picked out by the pointcut `SensitiveMethod`. This aspect is application-independent and must be specialized by the developer.

The first step of the weaving is to specialize the authorization aspect to the base model depicted in Fig. 8. Through a graphical weaving interface, the developer maps each abstract element of the aspect to its corresponding element in the base model. In this experiment, the developer maps the abstract method `SensitiveMethod` to the method `sendRequest` as shown in Fig. 10. After this step, the application-dependent aspect is automatically generated. Its specification is similar to the application-independent one except for the pointcut `SensitiveMethod` that will have the value `call(sendRequest)`.

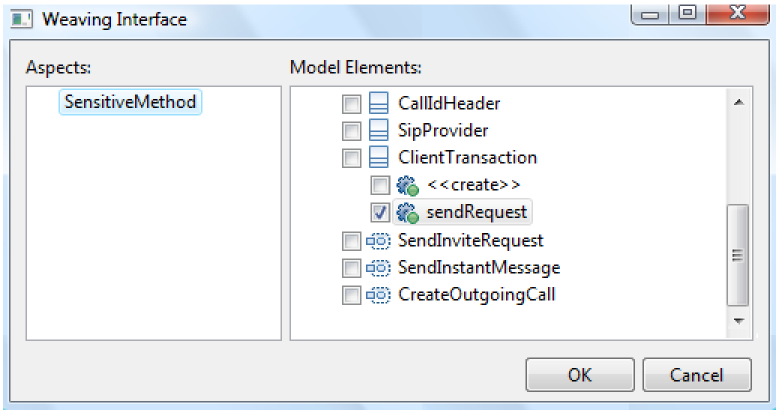


Fig. 10. Specialization of the Authorization Aspect

The next step of the weaving is the automatic identification of the join points where the check permission behavior shown in Fig. 9 should be injected. To achieve this, we first translate the textual expression of the pointcut `SensitiveMethod` to OCL. The resulting OCL expression is as follows:

self.oclIsTypeOf(CallOperationAction) and self.operation.name="sendRequest"

The evaluation of this OCL expression by the join point matching module returns as join points all the call operation actions that are invoking the method `sendRequest`. For instance, in the example of Fig. 8, the action `SendRequest` will be selected as matched join point. The last step of the weaving is the automatic injection of the check permission behavior into the base model at the identified join points. This is achieved by executing the QVT mapping rule that corresponds to the adaptation `AddCheckPermission` shown in Fig. 9. Finally, the resulting woven model for sending an instant message is generated as shown in Fig. 11.

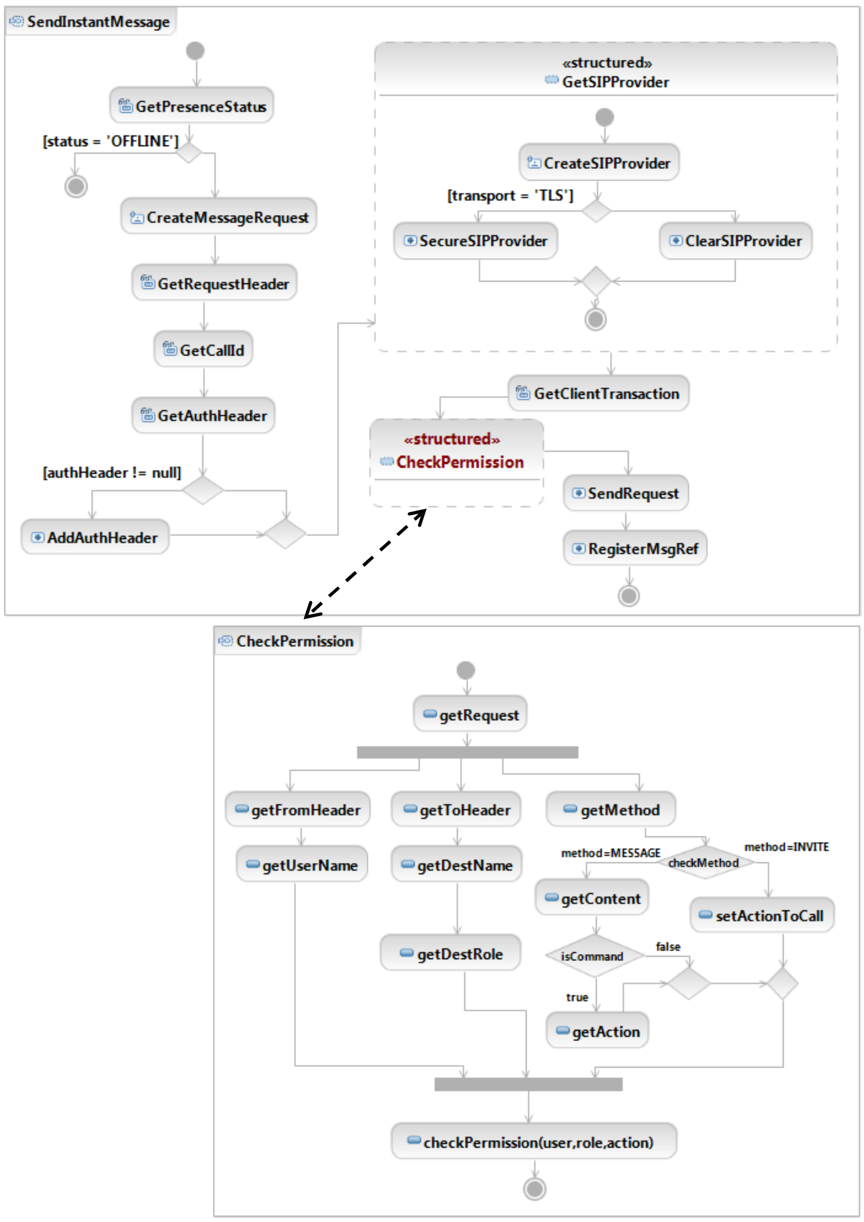


Fig. 11. Sending an Instant Message with Authorization - Woven Model

7 Related Work

Various practical approaches have been proposed for weaving aspects into different kinds of UML diagrams. In the following, we present an overview of these contributions.

The most relevant contribution is the one presented by Cui *et al.* [10] for modeling and integrating aspects with UML activity diagrams. Compared to this contribution that supports only adding new elements before and after the matched join points, our framework considers also replacing existing elements by new ones and removing elements. In addition, control nodes are also considered as join points. Algorithms for matching and weaving are provided in [10]. However, there is no formal semantics for these processes.

Fuentes and Sánchez [12] have proposed a model weaver for aspect-oriented executable UML models. Advice pieces are modeled as activity diagrams and injected into the base model as structured activities. Pointcuts are specified using sequence diagrams and intercept only observable behaviors such as sending and receiving of messages. This weaver supports adding new behaviors before, after, and around join points, but does not support removing behaviors. The weaving is performed on the XMI representation of the models using XSLT transformations. Therefore, the graphical representation of the woven model is not supported.

Zhang *et al.* [21] have presented Motorola WEAVR; a tool for weaving aspects into executable UML state machines. This weaver supports only two types of join points that are action and transition. It is implemented on top of the Telelogic TAU G2. Accordingly, it is tool-dependent and not portable.

Groher and Voelter [13] have presented XWeave; a weaver that supports the weaving of models and meta-models. This weaver is implemented following a model-to-model transformation approach using the openArchitectureWare framework. Compared to our framework, this weaver is limited only to the addition of new elements to the base model. It does not support removing or replacing existing elements.

Morin *et al.* have presented GeKo [17] that supports weaving of class, state machine, and sequence diagrams. The weaving is implemented as model transformations using Kermeta language [4]. The GeKo approach is based on the definition of mappings between the pointcut and the base model, and the pointcut and the advice. From these mappings, the elements to be added, deleted, or replaced are identified.

Whittle *et al.* [20] have developed MATA, a tool for modeling and composing UML models based on graph transformations. MATA supports weaving aspects into class, sequence, and state machine diagrams. In contrast to our approach, in MATA there are no explicit join points; any model element can be a join point. The weaving is implemented as graph transformation rules. Accordingly, both aspect and based models need to be transformed into graphs. After the weaving, the result is transformed back to a UML model.

Klein *et al.* [16] have proposed a semantic-based weaving algorithm for hierarchial message sequence charts, a formalism similar to UML sequence diagrams. The weaving algorithm is implemented as a set of transformations. The matching process consists of transforming the original model in such a way that pointcuts only match a finite number of paths. Similar to our approach, they support adding, replacing, and removing behaviors.

8 Conclusion and Future Work

We have presented in this paper a formal and practical framework for aspect weaving in UML activity diagrams. In this respect, syntax of activity diagrams together with their corresponding adaptations have been defined to express the matching and weaving semantics. Afterwards, we have proved the correctness and the completeness of the matching and weaving algorithms with respect to the defined semantics. At the end, we presented the methodology that we followed to implement the matching and weaving rules together with a case study that demonstrates the viability and the relevance of our framework. By adopting the standard OCL for evaluating the pointcuts, our approach is generic enough to specify a wide set of pointcut expressions covering various activity diagram elements. The adoption of the standard QVT for implementing the weaving rules extends portability of the designed weaver to all tools supporting QVT language beyond current implementation in RSA. As a future work, we plan to extend the weaving semantics to include other UML diagrams, such as, class diagrams, sequence diagrams, and state machine diagrams.

References

1. OCaml for Scientists (2010), `http://caml.inria.fr/pub/docs/manual-ocaml`
2. CUP: LALR Parser Generator for Java (2010), `http://www2.cs.tum.edu/projects/cup/`
3. Eclipse QVT Operational (2010), `http://www.eclipse.org/modeling/m2m/downloads/index.php?project=qvtoml`
4. Kermeta (2010), `http://www.kermeta.org/`
5. MOF Query/View/Transformation, Version 1.0. (2010), `http://www.omg.org/spec/QVT/1.0/`
6. Object Constraint Language, Version 2.2.(2010), `http://www.omg.org/spec/OCL/2.2/`
7. SIP Communicator Web site (2010), `http://sip-communicator.org/`
8. Unified Modeling Language (OMG UML): Superstructure, Version 2.2 (2010), `http://www.omg.org/spec/UML/2.2/Superstructure/PDF/`
9. Aspect-Oriented Modeling Workshop Web site (2010), `http://www.aspect-modeling.org/`
10. Cui, Z., Wang, L., Li, X., Xu, D.: Modeling and Integrating Aspects with UML Activity Diagrams. In: Shin, S.Y., Ossowski, S. (eds.) Proceedings of the Symposium on Applied Computing (SAC), pp. 430–437. ACM, New York (2009)
11. Fleurey, F., Baudry, B., France, R., Ghosh, S.: A Generic Approach for Automatic Model Composition. In: Proceedings of the Workshop on Aspect-Oriented Modeling, pp. 7–15. Springer, Heidelberg (2007)
12. Fuentes, L., Sánchez, P.: Designing and Weaving Aspect-Oriented Executable UML Models. Journal of Object Technology 6(7), 109–136 (2007)

13. Groher, I., Voelter, M.: XWeave: Models and Aspects in Concert. In: Proceedings of the Workshop on Aspect-Oriented Modeling, pp. 35–40. ACM, New York (2007)
14. IBM-Rational Software Modeler (2010), `http://www.ibm.com/software/awdtools/modeler/swmodeler/`
15. Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-Oriented Programming. In: Aksit, M., Auletta, V. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
16. Klein, J., Hélouët, L., Jézéquel, J.M.: Semantic-Based Weaving of Scenarios. In: Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD), pp. 27–38. ACM, New York (2006)
17. Morin, B., Klein, J., Barais, O., Jézéquel, J.: A Generic Weaver for Supporting Product Lines. In: Proceedings of the Workshop on Software Architectures and Mobility (EA), pp. 11–18. ACM, New York (2008)
18. Mouheb, D., Talhi, C., Nouh, M., Lima, V., Debbabi, M., Wang, L., Pourzandi, M.: Aspect-Oriented Modeling for Representing and Integrating Security Concerns in UML. In: Lee, R., Ormandjieva, O., Abran, A., Constantinides, C. (eds.) SERA 2010. SCI, vol. 296, pp. 197–213. Springer, Heidelberg (2010)
19. Sandhu, R., Ferraiolo, D., Kuhn, R.: The NIST Model for Role-Based Access Control: Towards A Unified Standard. In: Proc. of the ACM workshop on Role-Based Access Control, pp. 47–63 (2000)
20. Whittle, J., Jayaraman, P.: Mata: A Tool for AOM Based on Graph Transformation. In: Proceedings of the Aspect-Oriented Modeling Workshop, pp. 16–27. Springer, Heidelberg (2007)
21. Zhang, J., Cottenier, T., Berg, A., Gray, J.: Aspect Composition in the Motorola Aspect-Oriented Modeling Weaver. Journal of Object Technology. Special Issue on Aspect-Oriented Modeling 6(7), 89–108 (2007)

Using Temporal Logic for Dynamic Reconfigurations of Components

Julien Dormoy[1], Olga Kouchnarenko[1], and Arnaud Lanoix[2]

[1] University of Franche-Comté, Besançon, France
{jdormoy,okouchnarenko}@lifc.univ-fcomte.fr
[2] Nantes University, Nantes, France
arnaud.lanoix@univ-nantes.fr

Abstract. Dynamic reconfigurations increase the availability and the reliability of component-based systems by allowing their architectures to evolve at run-time. This paper deals with the formal specification and verification of dynamic reconfigurations of those systems using architectural constraints and temporal logic patterns.

The proposals of the paper are applied to the Fractal component model. Given a Fractal reference implementation of a component-based system, we specify its dynamic reconfigurations using a temporal pattern logic for Fractal, called FTPL, characterizing the correct behaviour of the system under some architectural constraints. We study system reconfigurations on which we verify these requirements, in particular by reusing the FPath and FScript tools.

1 Introduction

Component-based development provides significant advantages like portability, adaptability, re-usability, etc. when developing, e.g., Java Card smart card applications or when composing components or services within Service Component Architecture (SCA). The adaptability means that component-based systems must be adapted, or even adapt themselves [15] to their environments during their lifetime, and there is a need to support dynamic reconfigurations, including unanticipated ones [20]. To take up this challenge, this paper deals with the formal specification and verification of dynamic reconfigurations of component-based systems, and uses temporal patterns to monitor them.

The present paper makes the following contributions. The first contribution is a formal definition of the semantics of component-based systems with reconfigurations. To specify system reconfigurations, the second contribution is the definition of a linear time temporal logic based on architectural constraints which are first order configuration properties, and on event properties. For temporal operators, its expressive power is related to the well known linear time temporal logic (LTL) [19]. The third contribution is the application of the paper proposals to the Fractal component model [10]. For the Fractal component model, run-time verification issues are addressed to monitor reconfigurations during system lifetime.

More precisely, this paper follows the line of reasoning suggested in [12], where the system consistency during its dynamic reconfigurations relies on *integrity constraints*—predicates on assemblies of architectural elements and component states. To go further, we propose to support dynamic reconfigurations by using more complex architectural constraints and linear temporal logic patterns. These temporal patterns have been inspired by the pragmatic work of the SanTos Specification Pattern Project [13], and works on temporal extension of JML [21,9,14] helping Java programmers in writing formal specifications. We refer to this temporal extension as FTPL, for Temporal Pattern Language, prefixed by an 'F' to denote its adaptation to Fractal-like component systems and to first-order integrity constraints over them.

The proposals of the paper are applied to the Fractal component model. Given a Fractal reference implementation of a component-based system, we specify its dynamic reconfigurations using FTPL, characterizing the correct behaviour of the system under some architectural constraints. We monitor system reconfigurations by reusing the FPath and FScript tool supports.

The remainder of the paper is organised as follows. After giving a motivating example in Sect. 2, we formally define the semantics of component-based systems with reconfigurations in Sect. 3. To support system reconfigurations, Sect. 4 introduces a linear time temporal logic based on architectural constraints and events. Then, the proposals of the paper are applied to and illustrated on the Fractal component model in Sect. 5. Finally, Section 6 concludes before discussing related work.

2 Motivating Example

To motivate and to illustrate our approach, let us consider an example of a HTTP server from [11]. The architecture of this server is displayed in Fig. 1.

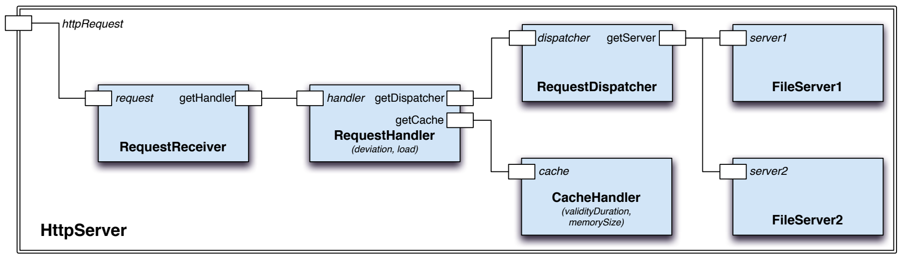


Fig. 1. HTTP Server architecture

The **RequestReceiver** component reads HTTP requests from the network and transmits them to the **RequestHandler** component. In order to perform a request, **RequestHandler** can either use the cache (with the component **CacheHandler**) or transmit the request to the **RequestDispatcher** component.

This component uses a set of file servers (like the **FileServer1** and **FileServer2** components) to answer the request.

This architecture provides a cache (**CacheHandler** component) and a load controller (**RequestDispatcher** component) in order to control the response time of the HTTP server. To keep the response time as short as possible whatever the number of requests is, in [11] the authors propose to dynamically reconfigure the HTTP server. For that, some requirements have been identified:

1. The **CacheHandler** component is used only if the number of similar HTTP requests is high.
2. The quantity of allocated memory for the **CacheHandler** component must depend on the overall load of the server.
3. The validity of data in the cache must also depend on the overall load of the server.
4. The number of used file servers depends on the overall load of the server.

In order to take these requirements into account, the **RequestHandler** and **CacheHandler** components are equipped with some parameters. The number of requests (load) and the percentage of similar requests (deviation) are two parameters defined for the **RequestHandler** component. The memory size (memorySize) and the data validity duration (validityDuration) are two parameters defined for the **CacheHandler** component.

We consider that the HTTP server can be reconfigured during the execution by the following reconfigurations

1. AddCacheHandler and RemoveCacheHandler which are respectively used to add and remove the **CacheHandler** component,
2. AddFileServer and removeFileServer which are respectively used to add and remove the **FileServer2** component,
3. MemorySizeUp and MemorySizeDown which are respectively used to increase and to decrease the MemorySize value,
4. DurationValidityUp and DurationValidityDown to respectively increase and decrease the ValidityDuration value.

3 Architectural (Re-)Configuration Model

This section gives means for specifying component-based systems with reconfigurations. A model we propose here is inspired by the model in [16,18] given for Fractal. Both models are graphs allowing one to represent component-based architectures and reconfiguration operations and to reason about them. Unlike [16,18], in our model, only the basic and generic concepts are considered to allow their application to various hierarchical component models: *components* as runtime entities, *required* and *provided interfaces* as interaction points between components, *bindings* to link component interfaces. Components are either *primitive* components or *composite* components, and primitive components can have some attributes used as configuration parameters.

Basically, a component-based system with reconfigurations is characterized by a set of configurations and a set of actions that allow the modification of configurations.

3.1 Component-Based Architectures

In general, the system configuration is the specific definition of the elements that define or prescribe what a system is composed of. We define a configuration to be a set of architectural elements (components, interfaces and parameters) together with a relation to structure and to link them. We consider a graph-based representation as proposed in [16,18].

Definition 1. *A configuration c is a tuple $\langle Elem, Rel\rangle$ where:*

- *$Elem$ is a set of architectural elements, and*
- *$Rel \subseteq Elem \times Elem$ is a relation between architectural elements.*

In our model, the architectural elements are the core entities of a component-based system: components, required or provided interfaces, and parameters.

Definition 2. *The set of architectural elements $Elem$ is defined by:*

$$Elem = Component \uplus Interface \uplus Parameter \uplus Type$$

where

- *$Component$ is a non-empty set of the core entities, i.e components;*
- *$Interface = Required \uplus Provided$ is a set of the (required and provided) interfaces;*
- *$Parameter$ is a set of component parameters;*
- *$Type$ is a set of data types associated with parameters.*

Each data type is a set of data values. For the sake of readability, we identify data type names with the corresponding data domains.

The architectural relation Rel then expresses various links between architectural elements. For example, it allows specifying that a component has an interface or a parameter, or that a component contains other (sub-)components, or that an interface is linked to another one.

Definition 3. *The architectural relation Rel is defined by:*

$$Rel = \begin{array}{l} ProvidedBy \uplus RequiredBy \uplus ParameterOf \uplus \\ TypeOf \uplus ValueOf \uplus ChildOf \uplus Binding \uplus Delegate \end{array}$$

where

- *$ProvidedBy : Provided \to Component$ is a total surjective function which gives the component having a provided interface;*
- *$RequiredBy : Required \to Component$ is a total function which gives the component with a required interface;*
- *$ParameterOf : Parameter \to Component$ is a total function which gives the component of a considered parameter;*

- $TypeOf : Parameter \to Type$ *is a total function which gives the type associated with a considered parameter;*
- $ValueOf : Parameter \to \bigcup_{type \in Type} type$ *such that* $\forall p \in Parameter : ValueOf(p) \in TypeOf(p)$, *is a total function which gives the current value of a considered parameter;*
- $ChildOf \subseteq Component \times Component$ *is a irreflexive and antisymetric relation linking composite components to their sub-components*[1] *such that:*
  - $\forall\ c, c' \in Component.\ ((c, c') \in ChildOf \Rightarrow \forall\ p \in Parameter.\ (ParameterOf(p) \neq c))$, *i.e, composite components have no parameter;*
  - *Let* $ChildOf^+$ *be the transitive closure of* $ChildOf$. *Then,* $\forall c, c' \in Component.\ ((c, c') \in ChildOf^+ \Rightarrow c \neq c')$, *i.e.,* $ChildOf$ *is an acyclic relation;*
- $Binding : Provided \to Required$ *is a partial function such that* $\forall\ ip \in Provided, ir \in Required.\ (Binding(ip) = ir \Rightarrow ProvidedBy(ip) \neq RequiredBy(ir) \wedge \exists\ c \in Component.\ ((c, ProvidedBy(ip)) \in ChildOf \wedge (c, RequiredBy(ir)) \in ChildOf))$, *i.e., two linked interfaces do not belong to the same component, but the corresponding components are sub-components of the same composite component;*
- $Delegate : Interface \to Interface$ *is a partial injective function to specify the delegation from a sub-component interface to the composite interface such that* $\forall\ i, i' \in Interface.\ (Delegate(i) = i' \Rightarrow (ProvidedBy(i'), ProvidedBy(i)) \in ChildOf \vee (RequiredBy(i'), RequiredBy(i)) \in ChildOf)$, *i.e., when delegating, the component providing* i *is a sub-component of the component providing* i', *or the component requiring* i *is a sub-component of the component requiring* i'.

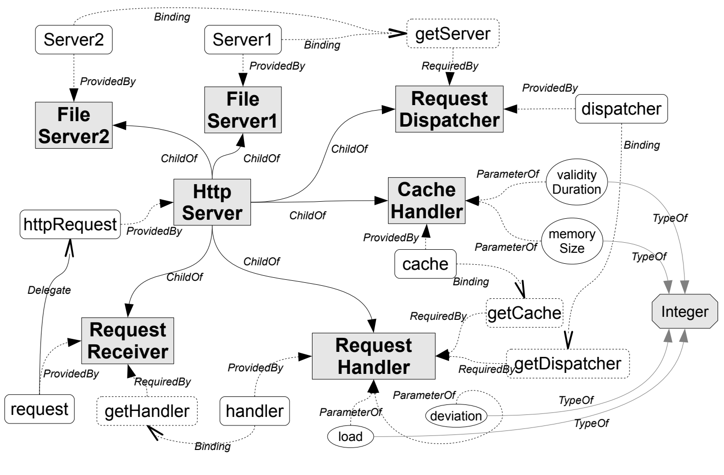


Fig. 2. Graph representation of the HTTP Server example

[1] For any $(p, q) \in ChildOf$, we say that p has a sub-component q, i.e. q is a child of p.

Example 1. Figure 2 illustrates main lines of Definition 3 on the example from Section 2. In this figure, the architectural elements are depicted as boxes and circles, whereas architectural relations are represented by arrows. For example, the request architectural element (at the bottom on the left) is an interface provided by **RequestReceiver**. The **RequestReceiver** architectural element is a subcomponent of the **HttpServer** composite component which provides the httpRequest interface. The request interface delegates results passing to the httpRequest interface.

3.2 Dynamicity of Component Architectures

To support system evolution, some component models provide mechanisms to dynamically reconfigure the component-based architecture, during their execution. These dynamic reconfigurations are then based on architectural modifications, among the following primitive operations:

– instantiation/destruction of components;
– addition/removal of components;
– binding/unbinding of component interfaces;
– starting/stopping components;
– setting parameter values of components;

or combinations of them. Notice that reconfigurations are not the only manner to make a component architecture evolve. The normal running of different components also changes the architecture by modifying parameter values or stopping components, like in the example.

Considering the component-based architecture model given in Sect. 3.1, an operation which makes the component architecture evolve by a reconfiguration action or by running, is modelled by a graph transformation operation adding or removing nodes and/or arcs in the graph of the configuration. An evolution operation op transforms a configuration $c = \langle Elem, Rel \rangle$ into another one $c' = \langle Elem', Rel' \rangle$. It is represented by a transition from c to c', noticed $c \overset{op}{\to} c'$. Among the evolution operations (running operations and reconfigurations), we particularly focus on the reconfiguration ones, which are either the above-mentioned primitive architectural operations or their compositions. The remaining running operations are all represented by a generic operation, called the *run* operation; it is also the case for sequences of running operations.

Definition 4. *The set of evolution operations $\mathcal{R}_{run}$ is defined by:*

$$\mathcal{R}_{run} = \mathcal{R} \cup \{run\}$$

with

– *$\mathcal{R}$ is a finite set of reconfiguration operations;*
– *run is an action renaming one or more running operations.*

Given a component architecture and the set $\mathcal{R}_{run}$ of reconfiguration operations, the behaviour of the component architecture is defined as a transition system labelled over $\mathcal{R}_{run}$.

Definition 5. *The evolution of a component architecture is defined by the transition system* $\langle \mathcal{C}, \mathcal{R}_{run}, \rightarrow \rangle$ *where:*

– $\mathcal{C} = \{c, c_1, c_2, \ldots\}$ *is a set of configurations,*
– $\mathcal{R}_{run}$ *is a finite set of evolution operations,*
– $\rightarrow \subseteq \mathcal{C} \times \mathcal{R}_{run} \times \mathcal{C}$ *is the reconfiguration relation.*

Given the evolution of a component architecture, we can now define the useful notions of path, trace, etc.

Definition 6. *Given the model* $M = \langle \mathcal{C}, \mathcal{R}_{run}, \rightarrow \rangle$*, an evolution path (or a path for short)* σ *of* M *is a (possibly infinite) sequence of configurations* $c_0, c_1, c_2, \ldots$ *such that* $\forall i \geq 0. \exists r_i \in \mathcal{R}_{run}. c_i \xrightarrow{r_i} c_{i+1} \in \rightarrow))$.

We use $\sigma(i)$ *to denote the i-th configuration of a path* σ*. The notation* σ_i *denotes the suffix path* $\sigma(i), \sigma(i+1), \ldots$*, and* σ_i^j *denotes the segment path* $\sigma(i), \sigma(i+1), \sigma(i+2), \ldots, \sigma(j-1), \sigma(j)$*. The segment path is infinite in length when the last state of the segment is repeated infinitely often.*

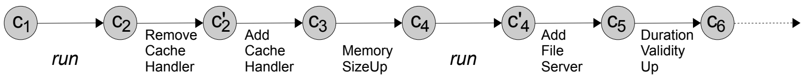


Fig. 3. Part of an evolution path of the HTTP server example

Example 2. A possible evolution path of the HTTP server is given in Fig. 3. In this path,

– c_1 is a configuration of the HTTP server without the **CacheHandler** and **FileServer2** components;
– c_2 is obtained from c_1: the load value was changed following the running of the **RequestHandler** component;
– c_2' is the same configuration as c_2. Without the **CacheHandler** component, the RemoveCacheHandler reconfiguration cannot terminate, it is then roll-backed without any modification;
– c_3 is obtained from the configuration c_2 by adding **CacheHandler**, following the AddCacheHandler reconfiguration operation;
– c_4 is the configuration c_3 in which the memorySize value was increased;
– c_4' is the same configuration as c_4. The result of the running is not observable;
– c_5 is obtained from c_4 by adding the **FileServer2** component;
– c_6 is like the configuration c_5 but the durationValidity value was increased.

4 Temporal Logic for Dynamic Reconfigurations

This section presents the syntax and the semantics of the temporal logic for dynamic reconfigurations. This logic, called FTPL, is inspired by the temporal logic in [21] designed to help Java programmers in writing formal specifications.

4.1 Syntax of the Logic

Let us first give the FTPL syntax:

<*TempProp*>	::=	**after** <*Events*> <*TempProp*>
	|	**before** <*Events*> <*TraceProp*>
	|	<*TraceProp*> **until** <*Events*>
	|	<*TraceProp*> **unless** <*Events*>
	|	**between** <*Events*> <*Events*> <*TraceProp*>
	|	<*TraceProp*>
<*TraceProp*>	::=	**always** ConfigProp
	|	**eventually** ConfigProp
	|	**never** ConfigProp
	|	<*TraceProp*> $\wedge$ <*TraceProp*>
	|	<*TraceProp*> $\vee$ <*TraceProp*>
<*Events*>	::=	<*Event*> , <*Events*>
	|	<*Event*>
<*Event*>	::=	ReconfigOp **called**
	|	ReconfigOp **normal**
	|	ReconfigOp **exceptional**
	|	ReconfigOp **terminates**

This language consists of different layers:

- the configurations properties,
- the reconfiguration operations,
- the trace properties,
- the temporal properties.

4.2 Semantics of FTPL

Let us now give the FTPL semantics. It is defined by induction on the form of the formulas.

Configuration properties. Basically, there is a need to express properties on the configurations, i.e constraints on the architectural elements and the relations between them. These constraints are specified using first order logic formulas, sets and relational operations on the primitive sets and relations given in Sect. 3.

Given the model M, we say that a configuration property cp is valid on a configuration $c = \langle Elem, Rel \rangle$, written $M, c \models cp$, when the evaluation of cp on the configuration $c = \langle Elem, Rel \rangle$ is true. When M is understood, we simply write $c \models cp$.

The configuration properties are expressed at different specification levels. At the component model level, the constraints are common to all the component architectures. In addition, some constraints must be expressed to restrict a family of component architectures (a profile level), or to restrict a specific component architecture (an application level).

Example 3. Let *CacheConnected* be a configuration property defined by

$$\exists \mathsf{cache}, \mathsf{getCache} \in Interface.(ProvidedBy(\mathsf{cache}) = \textbf{CacheHandler} \wedge RequiredBy(\mathsf{getCache}) = \textbf{RequestHandler} \wedge Binding(\mathsf{cache}) = \mathsf{getCache}).$$

It expresses that the component **CacheHandler** is connected to **RequestHandler**. For the evolution path from Fig. 3 we have: $c3 \models CacheConnected$ whereas $c2 \not\models CacheConnected.$

Event properties We want to observe reconfiguration action effects, for example when a reconfiguration is called or when it terminates, to specify and verify properties over them. Given a reconfiguration operation r in $\mathcal{R}$, we consider the following events:

– r **called** denoting that the reconfiguration r has been invoked,
– r **normal** denoting that the reconfiguration r has terminated normally,
– r **exceptional** denoting that the configuration r has rollbacked.

Definition 7. *Let σ be an evolution path in M, and r a reconfiguration operation in $\mathcal{R}$. Given an event property e, depicted <Event> in the FTPL syntax , its validity on the i-th configuration of σ, denoted $\sigma(i) \models e$, is inductively defined on the form of e by:*

$\sigma(i) \models r$ **called**	*iff* $\exists \sigma(i+1).(\sigma(i) \xrightarrow{r} \sigma(i+1) \in \rightarrow)$
$\sigma(i) \models r$ **normal**	*iff* $i > 0 \wedge \sigma(i-1) \models r$ **called** $\wedge\, \sigma(i-1) \neq \sigma(i)$
$\sigma(i) \models r$ **exceptional**	*iff* $i > 0 \wedge \sigma(i-1) \models r$ **called** $\wedge\, \sigma(i-1) = \sigma(i)$
$\sigma(i) \models r$ **terminates**	*iff* $\sigma(i) \models r$ **normal** $\vee$ $\sigma(i) \models r$ **exceptional**

Given an evolution path σ, and a list of event properties $E = e_1, \ldots, e_n$, depicted <*Events*> in the syntax, we say that E is valid on the i-th configuration of σ, denoted $\sigma(i) \models E$, iff at least one event of the list E is valid on $\sigma(i)$.

Example 4. Let us consider again the evolution path displayed in Fig. 3. We have: $c3 \models$ MemorySizeUp **called**, $c5 \models$ AddFileServer **normal** and $c2 \models$ RemoveCacheHandler **exceptional**.

Trace Properties A trace property expresses a constraint which must be true when the component-based architecture changes, i.e on the evolution path.

Definition 8. *Let σ be an evolution path, and cp a configuration property. Given a trace property trp, depicted <TraceProp> in the FTPL syntax, its validity on σ, denoted $\sigma \models trp$, is inductively defined on the form of trp by:*

$\sigma \models$ **always** cp	*iff* $\forall i.(i \geqslant 0 \Rightarrow \sigma(i) \models cp)$
$\sigma \models$ **eventually** cp	*iff* $\exists i.(i \geqslant 0 \wedge \sigma(i) \models cp)$
$\sigma \models trp_1 \wedge trp_2$	*iff* $\sigma \models trp_1 \wedge \sigma \models trp_2$
$\sigma \models trp_1 \vee trp_2$	*iff* $\sigma \models trp_1 \vee \sigma \models trp_2$

Intuitively,

- **always** *cp* is valid on an evolution path σ iff *cp* is valid on each configuration of σ;
- **eventually** *cp* is valid on an evolution path σ iff *cp* is valid on one configuration of σ, at least;
- the semantics of conjunction and disjunction is classical.

In the syntax a keyword **never** is introduced: **never** *cp* being an abbreviation for **always** $\neg$ *cp*.

Let us remark that architectural invariants as presented in [16,18], can be handled within the FTPL framework by using **always** *cp*, where *cp* represents the considered architectural invariant.

Temporal Properties. Temporal properties are based on all the properties above, i.e. they exploit architectural constraints, event properties and trace properties, together with some temporal patterns, like in [21]. Let us recall that the SanTos Specification Pattern Project [13] has identified these temporal patterns as useful in practice.

Definition 9. *Let σ be an evolution path, E, E_1 and E_2 be lists of events, trp a trace property. Given a temporal property tpp, depicted <TempProp> in the FTPL syntax, its validity on σ, denoted $\sigma \models tpp$, is inductively defined on the form of tpp by:*

$$\boxed{\begin{array}{lll}
\sigma \models \textbf{after}\ E\ tpp & \textit{iff} & \forall i.(i \geqslant 0 \wedge \sigma(i) \models E \Rightarrow \sigma_i \models tpp) \\
\sigma \models \textbf{before}\ E\ trp & \textit{iff} & \forall i.(i > 0 \wedge \sigma(i) \models E \Rightarrow \sigma_0^{i-1} \models trp) \\
\sigma \models trp\ \textbf{until}\ E & \textit{iff} & \exists i.(i > 0 \wedge \sigma(i) \models E \wedge \sigma_0^{i-1} \models trp) \\
\sigma \models trp\ \textbf{unless}\ E & \textit{iff} & \forall i.(i \geqslant 0 \wedge \sigma(i) \not\models E \Rightarrow \sigma \models trp) \\
 & & \vee\ \exists i.(i \geqslant 0 \wedge \sigma(i) \models E \wedge \sigma_0^{i-1} \models trp)
\end{array}}$$

Intuitively,

- the property **after** *E tpp* is valid on an evolution path σ iff the validity of the event property E on a configuration of σ implies the validity of the temporal property *tpp* on the suffix of σ starting at this configuration;
- **before** *E trp* is valid on an evolution path σ iff for each configuration of σ, the validity of E on it means that the trace property *trp* is valid on the prefix of σ before the considered configuration;
- *trp* **until** E is valid on an evolution path σ iff there is a configuration of σ satisfying the event property E, and the trace property *trp* is valid on the prefix of σ ending before this event occurs;
- *trp* **unless** E is valid on an evolution path σ iff either the event property E is not valid on σ implying that the trace property *trp* is valid on σ, or there is a configuration of σ satisfying the event property E, and the trace property *trp* is valid on the prefix of σ before the corresponding event occurs;

A formula **between** E_1 E_2 *trp* has the same semantics as **after** E_1 (*trp* **until** E_2), i.e. the trace property *trp* is valid on the segment of σ consisting of the configurations in-between the configuration where E_1 holds (including it), and the configuration where E_2 holds (excluding it).

4.3 Application to the HTTP Server Example

Let us now illustrate the FTPL language use by expressing some properties for the example of HTTP server from Sect. 2.

Let us consider a temporal property saying that after the invocation of the reconfiguration operation AddCacheHandler, the **CacheHandler** component is always connected to **RequestHandler**, i.e. the *CacheConnected* configuration property from Example 3 holds on the path configurations after the invocation. This property is valid on the evolution path σ depicted in Fig. 3:

$$\sigma \models \textbf{after}\ \mathsf{AddCacheHandler}\ \textbf{called always}\ CacheConnected.$$

The following property expresses an architectural constraint saying that at least there is always one file server component connected.

$$\textbf{always}\ \big(\exists \mathsf{getServer} \in Interface.(RequiredBy(\mathsf{getServer}) = \\ \textsf{\textbf{RequestDispatcher}} \wedge \exists i \in Interface.Binding(i) = \mathsf{getServer})\big)$$

Let us now specify that the deviation must always be lower than 50 until the AddCacheHandler reconfiguration operation terminates normally:

$$(\textbf{always} \\ \exists deviation \in Parameter.(ParameterOf(deviation) = RequestHandler \\ \wedge\ deviation < 50))\ \textbf{until}\ \texttt{AddCacheHandler}\ \textbf{normal}$$

The following property says that between the exceptional termination of either the `MemorySizeUp` reconfiguration or the `DurationValidityUp` reconfiguration, and the normal termination of the `AddCacheHandler` reconfiguration operation, the number of used file servers is greater than 1:

$$\textbf{between}\ \big(\texttt{MemorySizeUp}\ \textbf{exceptional},\ \texttt{DurationValidityUp}\ \textbf{exceptional}\big) \\ \big(\texttt{addCacheHandler}\ \textbf{normal}\big) \\ \big(\exists \mathsf{getServer} \in Interface.(RequiredBy(\mathsf{getServer}) = \textsf{\textbf{RequestDispatcher}} \wedge \\ \exists i, i' \in Interface.(i \neq i' \wedge Binding(i) = \mathsf{getServer} \wedge Binding(i') = \mathsf{getServer})\big)$$

These examples show that architectural invariants and properties on immediate predecessors or target configurations of reconfiguration operations can be expressed by FTPL formulas. Further, they show that FTPL is more expressive than the proposals in [12]. Indeed, FTPL allows expressing event properties and temporal properties involving different kinds of properties satisfying temporal patterns which have been shown useful for practical applications.

4.4 On the Expressiveness of FTPL

Before considering FTPL temporal properties, let us recall that configuration properties are first order logic formulas.

Let us now consider temporal patterns. As explained in Sect.4, FTPL has been inspired by proposals in [13], and works on a temporal extension of JML [21,9,14], called JTPL. The semantics of JTPL temporal formulas and translation rules into JML annotations are detailed in [21] for *safety properties* and in [9] for *liveness properties*.

Let $\text{LTL}_k()$ denote a function translating the FTPL properties of the kind k into LTL properties. We adapt the above-mentioned works and propose the following translation of FTPL patterns into LTL formulas. In this translation *cp* is a configuration property, *trp*, trp_1 and trp_2 are trace properties, E, E_1 and E_2 are lists of event properties, and *tpp* is a temporal property. In FTPL, there is a way to decide whether a list of event properties is valid on a configuration or not. The following functions suppose that the same decision procedure exists in LTL.

Leaving aside the FTPL and LTL models, it is easy to see that FTPL trace properties can be rewritten in LTL as follows:

$\text{LTL}_{Trace}(\textbf{always}\ cp)$	$\text{G}(cp)$
$\text{LTL}_{Trace}(\textbf{eventually}\ cp)$	$\text{F}(cp)$
$\text{LTL}_{Trace}(trp_1\ \&\ trp_2)$	$\text{LTL}_{Trace}(trp_1) \wedge \text{LTL}_{Trace}(trp_2)$
$\text{LTL}_{Trace}(trp_1 \mid trp_2)$	$\text{LTL}_{Trace}(trp_1) \vee \text{LTL}_{Trace}(trp_2)$

For example, if in LTL atomic properties could be configuration properties, the safety property specifying that always there is at least one file server component connected, would be written in LTL as follows:

$$\text{G}(\exists \mathsf{getServer} \in Interface.(RequiredBy(\mathsf{getServer}) = \\ \textbf{\textsf{RequestDispatcher}} \wedge \exists i \in Interface.Binding(i) = \mathsf{getServer}))$$

For temporal properties, we have:

$\text{LTL}_{Temp}(\textbf{after}\ E\ tpp)$	$\text{G}(E \Rightarrow \text{LTL}_{Temp}(tpp))$
$\text{LTL}_{Temp}(\textbf{after}\ E\ trp)$	$\text{G}(E \Rightarrow \text{LTL}_{Trace}(trp))$
$\text{LTL}_{Temp}(\textbf{before}\ E\ trp)$	$\text{F}(E) \Rightarrow \text{LTL}_{Trace_B}(E, trp)$
$\text{LTL}_{Temp}(trp\ \textbf{until}\ E)$	$\text{F}(E) \wedge \text{LTL}_{Trace_B}(E, trp)$
$\text{LTL}_{Temp}(trp\ \textbf{unless}\ E)$	$\text{LTL}_{Trace_C}(E, trp)$
$\text{LTL}_{Temp}(\textbf{between}\ E_1\ E_2\ trp)$	$\text{LTL}_{Temp}(\textbf{after}\ E_1\ (trp\ \textbf{until}\ E_2))$
$\text{LTL}_{Temp}(trp)$	$\text{LTL}_{Trace}(trp)$

where:

$\text{LTL}_{Trace_B}(E, \textbf{always}\ cp)$	$cp\ \mathsf{U}\ E$
$\text{LTL}_{Trace_B}(E, \textbf{eventually}\ cp)$	$\neg(\neg cp\ \mathsf{U}\ E)$
$\text{LTL}_{Trace_C}(E, \textbf{always}\ cp)$	$\text{G}(cp) \vee (cp\ \mathsf{U}\ E)$
$\text{LTL}_{Trace_C}(E, \textbf{eventually}\ cp)$	$\text{F}(cp) \wedge \neg(\neg P\ \mathsf{U}\ E)$

Remark that a trace property is translated into LTL according to the temporal context in which the property is used, that is why we define two auxiliary functions LTL_{Trace_B} and LTL_{Trace_C}. The LTL_{Trace} function translates a trace property which either does not depend on a temporal property, or is inside an **after** temporal property. The LTL_{Trace_B} function is used to translate a trace property which is inside a **before** or an **until** temporal properties. Finally, the LTL_{Trace_C} function translates a trace property bounded by a **unless** temporal property.

For example, the property specifying that the deviation must always be lower than 50 until the `AddCacheHandler` reconfiguration operation terminates normally can be written in LTL as follows:

$$\mathrm{F}(\texttt{AddCacheHandler}\ \mathbf{normal}) \wedge (\\ \exists deviation \in Parameter.(ParameterOf(deviation) = RequestHandler \\ \wedge\ deviation < 50)\ \mathsf{U}\ \texttt{AddCacheHandler}\ \mathbf{normal})$$

5 Application to the Fractal Component Model

The Fractal component model [10] is one of the motivations of the present work because of its native support for dynamic architectures. Fractal also provides means for introspection and reconfigurations. Existing implementations for Fractal and its extensions offer a framework to experiment with FTPL-based reconfigurations. This section briefly describes some Fractal features and the existing language support for reconfigurations, before reporting on our experiments.

5.1 Overview of Fractal, FPath and FScript

The Fractal model is a hierarchical and reflective component model intended to implement, deploy and manage software systems [10]. A Fractal component is both a design and a run-time entity that consists of a unit of encapsulation, composition and configuration. A component is wrapped in a membrane which can show and control a casually connected representation of its encapsulated content. This content is either directly an implementation in case of a primitive component, or sub-components for composite components.

FPath [12] is a domain-specific language inspired by the XPath language that provides a notation and introspection mechanisms to navigate inside Fractal architectures. FPath expressions use the properties of components (e.g. the value of a component attribute or the state of a component) or architectural relations between components (e.g. the subcomponents of a composite component) to express queries about Fractal architectures.

FScript [12] is a language that allows the definition of reconfigurations of Fractal architectures. FScript integrates FPath seamlessly in its syntax, FPath queries being used to select the elements to reconfigure. To ensure the reliability of its reconfigurations, FScript considers them as transactions and integrates a back-end that implements this semantics on top of the Fractal model.

5.2 From the FTPL Model to Fractal

As explained above, the architectural model presented in Sec. 3 has been developed to capture the Fractal component model, among other component-based models with reconfigurations, like CCM, GCM, etc. To illustrate our proposals, in this section we give a part of the Http Server example encoded using our model (Fig. 4) as well as its implementation in FractalADL (Fig. 5).

$Component$	= { **HttpServer**, **RequestReceiver**, **RequestHandler**, . . . }
$Required$	= { getHandler, getDispatcher, getCache, . . .}
$Provided$	= { httpRequest, request, handler, . . .}
$Parameter$	= { load, deviation, . . .}
$Type$	= { Int}
$ProvidedBy$	= { httpRequest $\mapsto$ **HttpServer**, request $\mapsto$ **RequestReceiver**,
	handler $\mapsto$ **RequestHandler**, . . .}
$RequiredBy$	= { getHandler $\mapsto$ **RequestReceiver**,
	getDispatcher $\mapsto$ **RequestHandler**,
	getCache $\mapsto$ **RequestHandler**, . . .}
$ParameterOf$	= { load $\mapsto$ **RequestHandler**, deviation $\mapsto$ **RequestHandler**, . . .}
$TypeOf$	= { load $\mapsto$ Int, deviation $\mapsto$ Int, . . .}
$ValueOf$	= { load $\mapsto$ 100, deviation $\mapsto$ 50, . . . }
$ChildOf$	= { (**HttpServer**, **RequestReceiver**),
	(**HttpServer**, **RequestHandler**), . . . }
$Binding$	= { getHandler $\mapsto$ handler, . . .}
$Delegate$	= { request $\mapsto$ httpRequest }

Fig. 4. HttpServer example using Definitions 2 and 3

Let us recall that FractalADL[2] is the architecture description language for Fractal which allows implementing the Fractal component model.

We then use the FScript language to specify the reconfiguration operations presented in Sec. 3, and the FScript tool support to execute them. FScript is focused on the manipulation of architectural concepts and provides complete control of the architecture of the systems modeled in Fractal. Concretely, FScript takes an architecture of a current Fractal configuration and dynamically changes it according to a FScript file in order to create a new target architecture. The FScript implementation features guarantee that FScript reconfigurations always terminate and keep the system in a consistent and usable state.

For example, we specify the AddCacheHandler reconfiguration in FScript as presented in Fig. 6. This reconfiguration consists in creating a new instance of **CacheHandler** (`name`) and in specifying its name (`set-name`). Then, the component is integrated into the architecture (`add`) and the binding with the component **RequestHandler** is set (`bind`). Finally, the component **CacheHandler** is started (`start`).

5.3 Dynamic Verification

We now report on our experiments evaluating the feasibility of a run-time monitoring of FTPL properties. The monitoring on the execution of the architectural reconfiguration model depends on the property to be verified; It is either the satisfiability of a configuration property on one configuration, or the satisfiability of a temporal property on a sequence of component-based system architectures.

Verification of configuration properties. We use the FPath language support to verify a configuration property on one configuration. Indeed, any first order logic formula specifying a configuration property can be translated into an FPath expression, the FPath language having the same expressive power [16].

[2] `http://fractal.ow2.org/fractaladl/`

```
<definition name="HttpServer">
    <interface name="httpRequest" role="server"
        signature="java.lang.Runnable"/>
    <component name="RequestReceiver">
        <interface name="request" role="server"
            signature="java.lang.Runnable"/>
        <interface name="getHandler" role="client"
            signature="Handler"/>
        <content class="RequestReceiverImpl"/>
    </component>
    <component name="RequestHandler">
        <interface name="handler" role="server"
            signature="Handler"/>
        <content class="RequestHandlerImpl"/>
        <attributes signature="RequestHandlerAttributes">
            <attribute name="load" value="100"/>
            <attribute name="deviation" value="50"/>
        </attributes>
        <controller desc="primitive"/>
    </component>
    ...
    <binding client="this.httpRequest"
        server="RequestReceiver.request"/>
    <binding client="RequestReceiver.getHandler"
        server="RequestHandler.handler"/>
    ...
</definition>
```

Fig. 5. HTTP Server example in FractalADL

```
action addCache(root)
{
    newCache = new("CacheHandler");
    set-name($newCache, "CacheHandler");
    add($root, $newCache);
    bind($root/child::RequestHandler/interface::getcache, $newCache/
        interface::cache);
    start($newCache);
}
```

Fig. 6. AddCacheHandler Reconfiguration specified in FScript

For example, the *CacheConnected* configuration property from Example 3 can be expressed in FPath by:

```
$HttpServer/child::RequestHandler/interface::getCache/binding::cache/
    ↪ component::CacheHandler
```

Verification of temporal properties. Once the configuration properties are handled thanks to FPath, there is a need to deal with FTPL temporal properties. In [7], it is shown that the monitoring works well on specific safety properties. In FTPL the safety properties are properties containing only the keywords **after**, **before**, **unless** and **always**; The safety properties are also properties containing the **eventually** keyword iff they contain the **before** keyword to bound the **eventually** part. The other properties are liveness properties.

We have studied the feasibility of the safety properties monitoring by developing a controller in Fractal. This controller supervises the properties of interest each time a reconfiguration operation occurs. It retains the configurations appearing during the system execution to build a history and to use it for verification purpose. In order to make the monitoring easier, the controller divides the property into sub-properties and keeps each sub-property until it manages to validate it. For example, for the property **after** AddCacheHandler **called always** *CacheConnected*, the controller tries first to find a configuration where the AddCacheHandler **called** event property holds. Once such a configuration is found, the controller continues with the monitoring of the *CacheConnected* configuration property for all the following configurations.

The FTPL properties can be divided into two classes: the properties dealing with the past and the properties dealing with the future. As it is possible to determine what has happened in the past thanks to the history built by the controller, all the past properties can be monitored. But, due to the run-time verification, there are properties that cannot be ensured by the controller. For any property about the future containing **always**, the controller only ensures that the current configuration does not violate the property at the moment. For any property about the future containing **eventually** key-word, the controller cannot conclude neither.

As explained before, the controller is able to monitor the safety properties about the past, where as for the properties about the future, it only ensures their non-violation until the current configuration. An LTL-derived logic is proposed in [8] to specify and verify that a system behaviour will "presumably" conform or violate the property in the future. In [7], the monitoring of safety properties necessitates time intervals bounded in the past as well as in the future. To go further within the Fractal-based approach, and to handle liveness properties, a solution would be to exploit a variant specification of the **Loop** clause [14], to which liveness properties can be reduced.

6 Conclusion

In this paper we have developed a theoretical framework for dynamic reconfigurations of component-based systems. As a calculus for expressing and analysing reconfiguration and integrity constraints, we have utilised linear temporal logic, since formulas are interpreted over configuration sequences which naturally represent dynamic behaviour of component-based systems. For the Fractal component model, we have studied the feasibility of monitoring dynamic reconfigurations during system lifetime.

Related work. Dynamic reconfiguration of distributed applications is an active research topic [1,2,7,18] motivated by practical applications like those modelled in Fractal [10]. In the context of dynamic reconfigurations, ArchJava [3] gives means to reconfigure Java architectures, and the ArchJava language guarantees communication integrity at run-time. Barringer and al. also give a formal temporal logic based framework to reason about the evolution of systems [5].

The idea of using graph-based models to specify dynamic reconfigurations is not new at all. In [4], a temporal logic is proposed to specify and verify properties on graph transformation systems. In the Fractal-based framework, in [17,18] the authors have defined integrity constraints on a graph-based representation of Fractal, to specify the reliability of component-based systems. There are tools to allow the user to ensure the reliability of those reconfigurations at run-time.

Our model in Sect. 3 is closely related to the model proposed in [18] for the Fractal component systems but unlike [18], our model lays down only general architectural constraints. In this sense it can be considered as a generalisation of the Fractal-oriented model. Moreover, our model seems to be general enough to give operational semantics to other component-based systems. On the integrity and architectural constraint side, the FTPL logic allows us to specify architectural constraints more complex that only architectural invariants in [12].

Among other applications, our proposals aims at an active monitoring of component-based systems. The active monitoring involves interpreting a configuration data set and acting on those data to (re-)configure the system accordingly. This may simply be a validation of the target configuration, or a reconfiguration operation interruption. In [7,6], Basin and al. have shown the feasibility of monitoring temporal (safety) properties and, more recently, security properties using a runtime monitoring approach for metric First-order temporal logic (MFOTL). The semantics of MFOTL has been defined with respect to timed temporal structures. Like the model in [7], our model is a first-order structure, but instead of considering a sequence of time stamps, we focus on reconfiguration operations. Although our main motivation and hence the model are different, their algorithms for monitoring temporal safety properties would be adapted for performing dynamic reconfigurations of component-based systems.

References

1. Aguilar Cornejo, M., Garavel, H., Mateescu, R., De Palma, N.: Specification and Verification of a Dynamic Reconfiguration Protocol for Agent-Based Applications. Research Report RR-4222, INRIA (2001)
2. Aguirre, N., Maibaum, T.: A temporal logic approach to the specification of reconfigurable component-based systems. Automated Software Engineering (2002)
3. Aldric, J.: Using types to enforce architectural structure. In: WICSA 2008, pp. 23–34 (February 2008)
4. Baldan, P., Corradini, A., König, B., Lluch Lafuente, A.: A temporal graph logic for verification of graph transformation systems. In: Fiadeiro, J.L., Schobbens, P.-Y. (eds.) WADT 2006. LNCS, vol. 4409, pp. 1–20. Springer, Heidelberg (2007)
5. Barringer, H., Gabbay, D.M., Rydeheard, D.E.: From runtime verification to evolvable systems. In: Sokolsky, O., Taşıran, S. (eds.) RV 2007. LNCS, vol. 4839, pp. 97–110. Springer, Heidelberg (2007)
6. Basin, D.A., Klaedtke, F., Müller, S.: Policy monitoring in first-order temporal logic. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 1–18. Springer, Heidelberg (2010)
7. Basin, D.A., Klaedtke, F., Müller, S., Pfitzmann, B.: Runtime monitoring of metric first-order temporal properties. In: IARCS, FSTTCS 2008, India, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik. LIPIcs, vol. 2, pp. 49–60 (2008)

8. Bauer, A., Leucker, M., Schallhart, C.: Comparing LTL Semantics for Runtime Verification. Journal of Logic and Computation, JLC (2010)
9. Bellegarde, F., Groslambert, J., Huisman, M., Julliand, J., Kouchnarenko, O.: Verification of liveness properties with JML. Technical report RR-5331, INRIA (2004)
10. Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., Stefani, J.-B.: The fractal component model and its support in java. Softw., Pract. Exper. 36(11-12), 1257–1284 (2006)
11. Chauvel, F., Barais, O., Plouzeau, N., Borne, I., Jézéquel, J.-M.: Composition et expression qualitative de politiques d'adaptation pour les composants Fractal. In: GDR GPL 2009, Toulouse, France (January 2009)
12. David, P.-C., Ledoux, T., Léger, M., Coupaye, T.: FPath and FScript: Language support for navigation and reliable reconfiguration of Fractal architectures. Annales des Télécommunications 64(1-2), 45–63 (2009)
13. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: ICSE, pp. 411–420 (1999)
14. Giorgetti, A., Groslambert, J., Julliand, J., Kouchnarenko, O.: Verification of class liveness properties with java modelling language. In: IET Software (2008)
15. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer 36(1), 41–50 (2003)
16. Léger, M.: Fiabilité des Reconfigurations Dynamiques dans les Architectures à Composant. PhD thesis, Ecole Nationale Supérieure des Mines de Paris (2009)
17. Léger, M., Ledoux, T., Coupaye, T.: Reliable dynamic reconfigurations in the fractal component model. In: ARM 2007, pp. 1–6. ACM, New York (2007)
18. Léger, M., Ledoux, T., Coupaye, T.: Reliable dynamic reconfigurations in a reflective component model. In: Grunske, L., Reussner, R., Plasil, F. (eds.) CBSE 2010. LNCS, vol. 6092, pp. 74–92. Springer, Heidelberg (2010)
19. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems: Specification. Springer, Heidelberg (1992)
20. Redmond, B., Cahill, V.: Supporting unanticipated dynamic adaptation of application behaviour. In: Deng, T. (ed.) ECOOP 2002. LNCS, vol. 2374, pp. 205–230. Springer, Heidelberg (2002)
21. Trentelman, K., Huisman, M.: Extending jml specifications with temporal logic. In: Kirchner, H., Ringeissen, C. (eds.) AMAST 2002. LNCS, vol. 2422, pp. 334–348. Springer, Heidelberg (2002)

Modular Termination Analysis of Java Bytecode and Its Application to phoneME Core Libraries

D. Ramírez-Deantes[1], J. Correas[2], and G. Puebla[1]

[1] DLSIIS, Technical University of Madrid (UPM), Spain
[2] DSIC, Complutense University of Madrid (UCM), Spain

Abstract. Termination analysis has received considerable attention, traditionally in the context of declarative programming and, recently, also for imperative and Object Oriented (OO) languages. In fact, there exist termination analyzers for OO which are capable of proving termination of medium size applications by means of *global* analysis, in the sense that all the code used by such applications has to be proved terminating. However, global analysis has important weaknesses, such as its high memory requirements and its lack of efficiency, since often some parts of the code have to be analyzed over and over again, libraries being a paramount example of this. In this work we present how to extend the termination analysis in the COSTA system in order to make it modular by allowing separate analysis of individual methods. The proposed approach has been implemented. We report on its application to the termination analysis of the core libraries of the phoneME project, a well-known open source implementation of Java Micro Edition (JavaME), a realistic but reduced version of Java to be run on mobile phones and PDAs. We argue that such experiments are relevant, since handling libraries is known to be one of the most relevant open problems in analysis and verification of real-life applications. Our experimental results show that our proposal dramatically reduces the amount of code which needs to be handled in each analysis and that this allows proving termination of a good number of methods for which global analysis is unfeasible.

1 Introduction

It has been known since the pre-computer era that it is not possible to write a program which correctly decides, in all cases, if another program will *terminate*. However, termination analysis tools strive to find proofs of termination for as wide a class of (terminating) programs as possible. Automated techniques are typically based on analyses which track *size* information, such as the value of numeric data or array indexes, or the size of data structures. This information is used for specifying a *ranking function* which strictly decreases on a well-founded domain on each computation step, thus guaranteeing termination. In the last two decades, a variety of sophisticated termination analysis tools have been developed, primarily for less-widely used programming languages. These include analyzers for term rewrite systems [16], and logic and functional languages [18,9,17].

Termination-proving techniques are also emerging in the imperative paradigm [7,10,16] and the object oriented (OO for short) paradigm, where static analysis tools such as Julia [25], AProVE [21], and COSTA [1] are able to prove termination of non-trivial medium-size programs. In the context of OO languages, we focus on the problem of proving whether the execution of a method m terminates for any possible input value which satisfies m's precondition, if any. Solving this problem requires, at least in principle, a *global analysis*, since proving that the execution of m terminates requires proving termination of all methods transitively invoked during m's execution. In fact, the three analysis tools for OO code mentioned above require the code of all methods reachable from m to be available to the analyzer and aim at proving termination of all the code involved. Though this approach is valid for medium-size programs, we quickly get into scalability problems when trying to analyze larger programs. It is thus required to reach some degree of *compositionality* which allows decomposing the analysis of large programs into the analysis of smaller parts.

In this work we propose an approach to the termination analysis of large OO programs which is compositional and we (mostly) apply it by analyzing a method at a time. We refer to the latter as *modular*, i.e., which allows reasoning on a method at a time. Our approach provides several advantages: first, it allows the analysis of larger programs, since the analyzer does not need to have the complete code of the program nor the intermediate results of the analysis in memory. Second, methods are often used by several other methods. The analysis results of a shared method can be reused for multiple uses of the method.

The approach presented is flexible in the level of granularity: it can be used in a component-based system at the level of components. A specification can be generated for a component C by analyzing its code, and it can be deployed together with the component and used afterwards for analyzing other components that depend on this one. When analyzing a component-based application that uses C, the code of C does not need to be available at analysis time, since the specification generated can be used instead.

In order to evaluate the effectiveness of our approach, we have extended the COSTA analyzer to be able to perform modular termination analysis and we have applied the improved system to the analysis of the phoneME implementation of the core libraries of JavaME. Note that analysis of API libraries is quite challenging and a significant stress test for the analyzer for a number of reasons which are discussed in more detail in Section 5 below. The main contribution of this paper is that it provides a practical framework for the modular analysis of Java bytecode, illustrating its applicability to real programs by analyzing phoneME librares. These contributions are detailed from Section 4 onwards.

2 Non-modular Termination Analysis in COSTA

COSTA (see [4] and its references) is a cost [2] and termination [1] analyzer for Java bytecode. COSTA receives as input the signature of the method m whose termination (or cost) we want to infer. Method m is assumed to be available

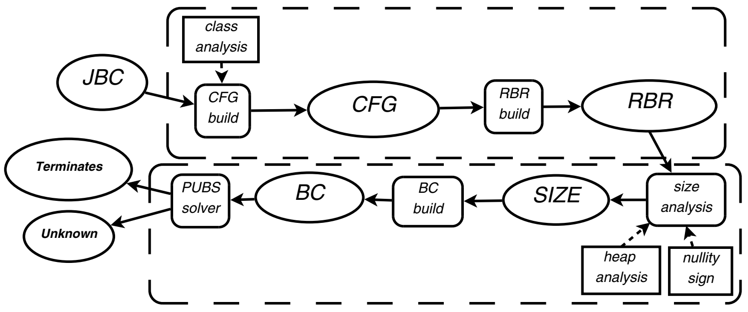


Fig. 1. Architecture of COSTA

in the classpath or default Java run-time environment (jre for short) libraries, together with all other methods and classes transitively invoked by m. Since there can be many more classes and methods in the classpath and jre than those reachable from m, a first step during analysis consists in identifying a set M of methods which includes all methods reachable from m. This phase is sometimes referred to as *program extraction* or *application extraction*. Then, COSTA performs a *global analysis* since, not only m, but all methods in the program M are analyzed.

We now briefly describe the overall architecture of COSTA, which is graphically represented in Figure 1. More details can be found in [3]. The dashed frames represent the two main phases of the analysis: (i) consists of extracting a program M from the method m plus the transformation of the bytecode for all methods in M into a suitable internal representation; and (ii) the actual static analysis. Input and output of the system are depicted on the left: by *JBC* we denote the bytecode of all classes in the classpath and jre plus the signature of a method and yields information about termination, indicated by *Terminates* (the analyzer has proved that the program terminates for all valid inputs) or *Unknown* (otherwise). Ellipses (e.g. *CFG*) represent *what* the system produces at each intermediate stage of the analysis; rounded boxes (e.g. "*CFG build*") indicate the *main steps* of the analysis process; square boxes (e.g. *class analysis*), which are connected to the main steps by dashed arrows, denote auxiliary analyses which allow obtaining more precise results. During the first phase, depicted in the upper half of the figure, the incoming *JBC* is transformed into a *rule-based representation* (*RBR*). In the second phase, depicted in the lower half of the figure, the system performs the actual termination analysis on the RBR.

2.1 From the Bytecode to the Rule-Based Representation

Generation of Control Flow Graphs Guided by Class Analysis. COSTA transforms the bytecode of a method into *Control Flow Graphs* (CFGs) by using techniques from compiler theory. As regards *Virtual invocation*, computing a precise approximation of the methods which can be executed at a given program point is not trivial. As customary in the analysis of OO languages, COSTA uses *class analysis* [24] (or points-to analysis) in order to precisely approximate this information. First, the CFG of the initial method is built, and class analysis is applied in order to approximate the possible runtime classes at each program point. This information is used to *resolve* virtual invocations. Methods which can be called at runtime are loaded, and their corresponding CFGs are constructed. Class analysis is applied to their body to include possibly more classes, and the process continues iteratively. Once a fixpoint is reached, it is guaranteed that all reachable methods have been loaded, and the corresponding CFGs have been generated.

As regards *exceptions*, COSTA handles internal exceptions (i.e., those associated to bytecodes as stated in the JVM specification), exceptions which are thrown (bytecode athrow) and possibly propagated back in methods, as well as finally clauses. Exceptions are handled by adding edges to the corresponding handlers. COSTA provides the options of ignoring only internal exceptions, all possible exceptions or considering them all.

Rule-Based Representation. Given a method m and its CFGs, a RBR for m is obtained by producing, for each basic block m_j in its CFGs, a rule which (1) contains the set of bytecode instructions within the basic block; (2) if there is a method invocation within the instructions, includes a call to the corresponding rule; and (3) at the end, contains a call to a *continuation rule* m_j^c which includes mutually exclusive rules to cover all possible continuations from the block. Note that several rules may be produced with the same name. A *procedure* P is the set of all rules with name P.

2.2 Context-Sensitive (Pre-)Analyses to Improve Accuracy

COSTA performs three context-sensitive analyses on the RBR based on abstract interpretation [12]: *nullity*, *sign* and *heap* analysis. These analyses improve the accuracy (and efficiency) of subsequent steps inferring information from individual bytecodes, and propagating it via a standard, top-down *fixpoint* computation.

Nullity Analysis aims at keeping track of reference variables which are definitely *null* or are definitely *non-null*. For instance, the bytecode $\mathsf{new}(s_i)$ allows assigning the abstract value *non-null* to s_i. The results of nullity analysis often allow removing rules corresponding to NullPointerException.

Sign Analysis aims at keeping track of the sign of variables. The abstract domain contains the elements $\geq$, $\leq$, $>$, $<$, $= 0$, $\neq 0$, $\top$ and $\bot$, partially ordered in a lattice. For instance, sign analysis of $\mathsf{const}(s_i, V)$ evaluates the integer value V and assigns the corresponding abstract value $= 0$, $>$ or $<$ to s_i, depending, resp., on if V is zero, positive or negative [12]. Knowing the sign of data allows removing RBR rules for arithmetic exceptions which are never thrown.

Heap Analysis obtains information related to variables and arguments located in the heap, a global data structure which contains objects (and arrays) allocated by the program. Infers properties like *constancy* and *cyclicity* of variables and arguments, and *sharing*, *reachability* and *aliasing* between variables and arguments in the heap [15]. They are used for inferring sound size relations on objects.

2.3 Size Analysis of Java Bytecode

From the RBR, *size* analysis takes care of inferring the relations between the values of variables at different points in the execution. To this end, the notion of *size measure* is crucial. The size of a piece of data at a given program point is an abstraction of the information it contains, which may be fundamental to prove termination. The COSTA system uses several size measures:

- *Integer-value* maps an integer value to its value (i.e., the size of an integer is the value itself). It is typically used in loops with an integer counter.
- *Path-length* [23] maps an object to the length of the maximum path reachable from it by dereferencing. This measure can be used to predict the behavior of loops which traverse linked data structures, such as lists and trees.
- *Array-length* maps an array to its length and is used to predict the behavior of loops which traverse arrays.

Size analysis works in two phases. In the first one, called *abstract compilation*, each bytecode, call or guard is *abstracted* by *linear constraints* on the size of its variables: for example, $\mathsf{iadd}(s_0, s_1, s_0')$ will be abstracted by the constraint $s_0'{=}s_1{+}s_0$, meaning that the size of s_0 after executing the instruction is the sum of the size of s_0 and s_1 before.

In the second phase, linear constraints replacing parts of the program can be propagated via a standard, bottom-up *fixpoint* computation, in order to combine the information about single rules. The goal of this global analysis is to have *size relations* on variables between the input of a rule (i.e., a block in the CFG) and another one which can be (directly or indirectly) called by the first one.

2.4 Inferring Termination

From the RBR and the results of size analysis, a set of *binary clauses* (*BC* in Figure 1) is produced, which capture calls among blocks together with information on how the values of variables change from one call to another. On such binary clauses, standard termination analysis techniques developed for i.e., termination of logic program can be applied. In particular, COSTA proves termination by using semantic-based techniques, relying on *binary unfolding* combined with *ranking functions*, as those in [9]. This is performed by means of the *PUBS* solver. More details on how termination proofs are performed in COSTA can be found in [1].

3 Abstract Interpretation Fundamentals

Before describing the modular analysis framework, a brief description to abstract interpretation is in order. *Abstract interpretation* [12] is a technique for static program analysis in which execution of the program is simulated on a description (or abstract) domain (D) which is simpler than the actual (or concrete) domain (C). Values in the description domain and sets of values in the actual domain are related via a pair of monotonic mappings $\langle\alpha, \gamma\rangle$: *abstraction* $\alpha : 2^C \to D$ and *concretization* $\gamma : D \to 2^C$ which form a Galois connection, i.e.

$$\forall x \in 2^C : \ \gamma(\alpha(x)) \supseteq x \quad \text{and} \quad \forall \lambda \in D : \ \alpha(\gamma(\lambda)) = \lambda.$$

The set of all possible descriptions represents a description domain D which is usually a complete lattice for which all ascending chains are finite. Note that in general $\sqsubseteq$ is induced by $\subseteq$ and α (in such a way that $\forall \lambda, \lambda' \in D : \ \lambda \sqsubseteq \lambda' \Leftrightarrow \gamma(\lambda) \subseteq \gamma(\lambda')$). Similarly, the operations of *least upper bound* ($\sqcup$) and *greatest lower bound* ($\sqcap$) mimic those of 2^C in some precise sense that depends on the particular abstract domain. A description $\lambda \in D$ *approximates* a set of concrete values $x \in 2^C$ if $\alpha(x) \sqsubseteq \lambda$. Correctness of abstract interpretation guarantees that the descriptions computed approximate all of the actual values which occur during the execution of the program.

In COSTA, abstract interpretation is performed on the rule based representation introduced in Section 2. We first introduce some notation. CP and AP stand for descriptions in the abstract domain. The expression $P{:}CP$ denotes a *call pattern*. This consists of a procedure P together with an entry pattern for that procedure. Similarly, $P \mapsto AP$ denotes an answer pattern, though it will be referred to as AP when it is associated to a call pattern $P{:}CP$ for the same procedure. Since a method is represented in the RBR as a set of interconnected procedures that start from a single particular procedure, the same notation will be used for methods: $m{:}CP$ denotes a call pattern that corresponds to an invocation to method m (i.e., the entry procedure for method m), and $m \mapsto AP$ denotes the answer pattern obtained after analyzing method m.

Context-sensitive abstract interpretation takes as input a program R and an initial call pattern $P{:}CP$, where P is a procedure and CP is a restriction of the values of arguments of P expressed as a description in the abstract domain D and computes a set of triples, denoted $\mathit{analysis}(R, P{:}CP) = \{P_1{:}CP_1 \mapsto AP_1, \ldots, P_n{:}CP_n \mapsto AP_n\}$. In each element $P_i{:}CP_i \mapsto AP_i$, P_i is a procedure and CP_i and AP_i are, respectively, the abstract call and answer patterns.

An analysis is said to be *polyvariant* if more than one triple $P{:}CP_1 \mapsto AP_1, \ldots, P{:}CP_n \mapsto AP_n$ $n \geq 0$ with $CP_i \neq CP_j$ for some i, j may be computed for the same procedure P, while a *monovariant* analysis computes (at most) a single triple $P{:}CP \mapsto AP$ for each procedure (with a call pattern CP general enough to cover all possible patterns that appear during the analysis of the program for P). Although in general context-sensitive, polyvariant analysis algorithms are more precise than those obtained with context-insensitive or monovariant analyses, monovariant algorithms are simpler and have smaller memory requirements. Context-insensitive analysis does not consider call pattern information, and therefore obtains as result of the analysis a set of pairs $\{P_1 \mapsto AP_1, \ldots, P_n \mapsto AP_n\}$, valid for any call pattern.

COSTA includes several abstract interpretation based analyses: nullity and sign are context-sensitive and monovariant, size is context-insensitive, and heap properties analysis [15] is context-sensitive and polyvariant.

4 Extending COSTA to Modular Termination Analysis

As described in Section 2, the termination analysis performed by COSTA is in fact a combination of different processes and analyses that receive as input a complete program and eventually produce a termination result. Our goal now is to obtain a *modular* analysis framework for COSTA which is able to produce termination proofs by analyzing programs one method at a time. I.e., in order to analyze a method m, we analyze the code of m only and (re-)use the analysis results previously produced for the methods invoked by m.

The communication mechanism used for this work is based on *assertions*, which store the analysis results for those methods which have already been analyzed. Assertions are stored by COSTA in a file per class basis and they keep information regarding the different analyses performed by COSTA: nullity, sign, size, heap properties, and termination.

Same as analysis results, assertions are of the form $m{:}Pre \mapsto Post$, where Pre is the *precondition* of the assertion and $Post$ is the *postcondition*. The precondition states for which call pattern the method has been analyzed. It includes information regarding all domains previously mentioned except size, which is context-insensitive. Pre_D (resp., $Post_D$) denotes the information of the precondition (resp., postcondition) related to analysis domain D. For example, $Pre_{nullity}$ corresponds to the information related to nullity in the precondition Pre. The postcondition of an assertion contains the analysis results for all domains produced after analyzing method m. Furthermore, the assertion also states whether COSTA has proved termination for that method.

In addition to assertions inferred by the analysis, COSTA has been extended to handle assertions written by the user, namely *assumed* assertions. These assertions are relevant for the cases in which analysis is not able to infer some information of interest that we know is correct. This can happen either because the analyzer is not precise enough or because the code of the method is not available to the analyzer, as happens with *native* methods, i.e., those implemented at low-level and for which no bytecode is available. The user can add assumed assertions with information for any domain. However, for the experiments described in Section 6 assumed assertions have been added manually for providing information about termination only, after checking that the library specification provided by Sun is consistent with the assertion. In assumed assertions where only termination information is available, abstract interpretation-based analyses take $\top$ as the postcondition for the corresponding methods.

4.1 Modular Bottom-Up Analysis

The analysis of a Java program using the modular analysis framework consists in analyzing each of the methods in the program, and eventually determining if the program will terminate or not for a given call pattern. Analyzing a method separately presents the difficulty that, from the analysis point of view, the code to be analyzed is *incomplete* in the sense that the code for methods invoked is not available. More precisely, during analysis of a method m there may be calls

$m'{:}CP$ and the code for m' is not available. Following the terminology in [14], we refer to determining the value of AP to be used for $m'{:}CP \mapsto AP$ as the *answer patterns problem.*

Several analysis domains existing in COSTA are context-sensitive, and all of them, except heap properties analysis, are monovariant. For simplicity, the modular analysis framework we present is monovariant as well. That means that at most one assertion $m{:}Pre \mapsto Post$ is stored for each method m. If there is an analysis result for m', $m'{:}Pre \mapsto Post$, such that CP is applicable, that is, $CP \sqsubseteq Pre_D$ in the domain D of interest, then $Post_D$ can be used as answer pattern for the call to method m' in m.

For applying this schema, it is necessary that all methods invoked by m have been analyzed already when analyzing method m. Therefore, the analysis must perform a bottom-up traversal of the call graph of the program. In order to obtain analysis information for m' which is applicable during the analysis of m, it is necessary to use a call pattern for m' in its precondition such that it is equal or more general than the pattern actually inferred during the analysis of m. We refer to this as the *call patterns problem.*

Solving the *Call* and *Answer Patterns Problems.* A possibility for solving the *call patterns problem* would be to make the modular analysis framework polyvariant: store all possible call patterns to methods in the program and then analyze those methods for each call pattern. This approach has two main disadvantages: on one hand, it is rather complex and inefficient, because all call patterns are stored and every method must be analyzed for all call patterns that appear in the program. On the other hand, it requires performing a fixpoint computation through the methods in the program instead of a single traversal of the call graph, since different call patterns for a method may generate new call patterns for other methods.

Another alternative is a context-insensitive analysis. All methods are analyzed using $\top$ as call pattern for all domains. In this approach, all assertions are therefore applicable, although in a number of cases $\top$ is too general as call pattern for some domains, and the information obtained is too imprecise.

The approach finally used in this work tries to find a balance between both approaches. A monovariant modular analysis framework simplifies a great deal the behavior of the modular analysis, since a single traversal of the call graph is required. In contrast, it is context-sensitive: instead of $\top$, a default call pattern is used, and the result of the analysis is obtained based on this pattern. This framework uses different values as call patterns, depending on the particular analysis being performed. The default call pattern for nullity and sign is $\top$. For Heap properties analysis, in cyclicity it is the pattern that indicates that no argument of the method is cyclic. For variable sharing, it is the one that states that no arguments share. The default call patterns used for analyzing methods are general enough to be applicable to most invocations used in the libraries and in user programs, solving the *call patterns problem.* However, there can be cases in which the call pattern of an invocation from other method is not included in the default pattern, i. e., $CP \not\sqsubseteq Pre_D$. If the code of the invoked method is available, COSTA will reanalyze it with respect to $CP \sqcup Pre_D$, even though it has been analyzed before for the default pattern. If the code is not available, $\top$ is used as answer pattern. A potential disadvantage of this approach is that all methods are analyzed with respect to a default call pattern, instead of the specific call pattern produced by the analysis. This means that the analyses in COSTA could produce more precise results when applied non modularly, even though they are monovariant, and it represents a possible loss of precision in the modular analysis framework. Nonetheless, in the experiments performed in Section 6 no method has been found for which it was not possible to prove termination using modular analysis, but it was proved in the non-modular model.

Cycles in the Call Graph. Analyzing just a method at a time and (re-)using analysis information while performing a bottom-up traversal of the call graph only works under the assumption that there are no cyclic dependencies among methods. In the case where there are strongly connected components (SCCs for short) consisting of more than one method, we can analyze all the methods in the corresponding SCC simultaneously. This presents no technical difficulties, since COSTA can analyze multiple methods at the same time. In some cases, we have found large cycles in the call graph that require analyzing many methods at the same time. In that case a different approach has been followed, as explained in Section 6. Therefore, in COSTA we perform a SCC study first to decide whether there are sets of methods which need to be handled as a unit.

Field-Sensitive Analysis. In some cases, termination of a method depends on the values of variables stored in the heap, i.e., fields. COSTA integrates a field-sensitive analysis [5] which, at least in principle, is a global analysis and requires that the source code of all the program be available to the analyzer. Nevertheless, in order to be able to use this analysis in the modular setting, a preliminary adaptation of that analysis has been performed. The field-sensitive analysis in COSTA is based on the analysis of program fragments named *scopes*, and modelling those fields whose behaviour can be reproducible using local variables. Fields must satisfy certain conditions in order to be handled as local variables. As a first step of the analysis, related scopes are analyzed in order to determine the fields that are consulted or modified in each scope. Given a method for which performing field-sensitive analysis is required in order to prove termination, an initial approximation to the set of methods that need to be analyzed together is provided by grouping those methods that use the same fields. We have precomputed these sets of methods by means of a non-modular analysis. Since the implementation of this preanalysis is preliminary and can be highly optimized, the corresponding time has not been included in the experiments in Section 6.

5 Application of Modular Analysis to phoneME Libraries

We have extended the implementation of COSTA for the modular analysis framework. In order to test its applicability, we have analyzed the core libraries of the phoneME project, a well-known open-source implementation of Java Micro Edition (JavaME). We now discuss the main difficulties associated to the analysis of libraries:

– *Entry points.* Whereas a self contained program has a single entry method (`main(String[])`), a library has many entry points that must be taken into account during the analysis.
– *It is designed to be used in many applications.* Each entry point must be analyzed with respect to a call pattern that represents any *valid* call from any program that might use it. By valid we mean that the call satisfies the precondition of the corresponding method.
– *Large code base.* A system library, especially in the case of Java, usually is a large set of classes that implement most of the features in the source language, leaving only a few specific functionalities to the underlying virtual machine, mainly for efficiency reasons or because they require low-level processing.
– *With many interdependencies.* It is usual that library classes are extensively used from within library code. As a result of this, library code contains a great number of interdependencies among the classes in the library. Thus, non-modular analysis of a library method often results in analyzing a large portion of the library code.
– *Implemented with efficiency in mind.* Another important feature of library code is that it is designed to be as efficient as possible. This means that readability and structured control flow is often sacrified for relatively small efficiency gains. Section 6 shows some examples in phoneME libraries.
– *Classes can be extended and methods overridden.* Using a library in a user program usually not only involves object creation and method invocation, but also library classes can be extended and library methods overridden.
– *Use of native code.* Finally, it is usual that a library contains calls to native methods, implemented in C or inside the virtual machine, and not available to the analyzer.

5.1 Some Further Improvements to COSTA

While trying to apply COSTA to the phoneME libraries, we have identified some problems which we discuss below, together with the solutions we have implemented. As mentioned above, our approach requires analyzing methods in reverse topological order of the call graph. For this purpose, we extended COSTA in order to produce the call graph of the program after transforming the bytecode to a CFG. The call graph shows the complex structure of the classes in phoneME libraries. Furthermore, apparently, some cycles among methods existed in some of the call graphs, mainly caused by virtual invocations. However, we observed that some potential cycles did not occur in practice. In these cases, either nullity and sign analyses remove some branches if they detect that are unreachable, or COSTA proves termination when solving the binary clauses system.

A few cases include a large cycle that involves many methods. Those cycles are formed by small cycles focused in few methods (basically from `Object`, `String` and `StringBuffer` classes), and a large cycle caused by virtual invocations from those methods. In order to speed up analysis, methods in small cycles have been analyzed at the same time, as mentioned above, and large cycles have been analyzed considering the modular, method at a time bottom up approach.

In addition, COSTA has been extended for a more refined control of which pieces of code we want to include or exclude from analysis. Now there are several *visibility* levels: `method`, `class`, `package`, `application`, and `all`. When `all` is selected, all related code is loaded and included in the RBR. In the other extreme, when `method` is selected only the current method is included in the RBR and only the corresponding assertions are available for other methods.

5.2 An Example of Modular Analysis of phoneME Libraries

As an example of the modular analysis framework presented in this paper, let us consider the method `Class.getResourceAsStream` in the phoneME libraries. It takes a string with the name of a resource in the application jar file and returns an object of type `InputStream` for reading from this resource, or `null` if no resource is found with that name in the jar file. Though COSTA analyzes bytecode, we show below the corresponding Java source for clarity of the presentation:

```
public java.io.InputStream getResourceAsStream(String name) {
    try {
        if (name.length() > 0 && name.charAt(0) == '/') {
           name = name.substring(1);
        } else {
           String clName = this.getName();
           int idx = clName.lastIndexOf('.');
           if (idx >= 0)
              name = clName.substring(0, idx+1).replace('.', '/') + name;
        }
        return new com.sun.cldc.io.ResourceInputStream(name);
    } catch (java.io.IOException x) { return null; }
}
```

In the source code of this method there are invocations to eleven methods of different classes (in addition to the eight methods explicitly invoked in the method code, the string concatenation operator in line 9 is translated to a creation of a fresh `StringBuffer` object and invocations to some of its methods.)

If the standard, non-modular approach of analysis is used, the analyzer would load the code of this method and all related methods invoked. In this case, there are 65 methods related to `getResourceAsStream`, from which 10 are native methods. In fact, using this approach COSTA is unable to prove termination. Using modular analysis, the call graph is traversed bottom-up, analyzing each method related to `getResourceAsStream` one by one. For example, the analysis of the methods invoked by `getResourceAsStream` has obtained the following information related to the nullity domain[1]:

Method	**call**	**result**
`StringBuffer.toString()`	n/a	`nonnull`
`StringBuffer.append(String)`	$\top$	`nonnull`
`StringBuffer.<init>()V`	n/a	n/a
`String.replace(char,char)`	$(\top,\top)$	`nonnull`
`com.sun.cldc.io.ResourceInputStream.<init>(String)`	`nonnull`	n/a
`String.substring(int)`	$\top$	`nonnull`
`String.length()`	n/a	$\top$
`String.substring(int,int)`	$(\top,\top)$	`nonnull`
`String.charAt(int)`	$\top$	$\top$

In this table, the call pattern refers to nullity information regarding the values of arguments and the result is related to the method return value. Despite of the call patterns generated by the analysis of `getResourceAsStream` shown above, when the bottom-up modular analysis computation is performed, all methods are analyzed with respect to the default call pattern $\top$. The analysis of `getResourceAsStream` uses the results obtained for those methods to generate the nullity analysis results for `getResourceAsStream`. The same mechanism is used for other domains: sign, size and heap related properties.

Finally, two native methods are invoked from `getResourceAsStream` (`lastIndexOf` and `getName`) that require assumed assertions. In this case, $\top$ is assumed as the answer pattern for those invocations.

5.3 Contracts for Method Overriding

As mentioned above, one of the most important features of libraries in OO languages is that classes can be extended by users at any point in time, including the possibility of overriding methods. This poses significant problems to modular static analysis, since classes and methods which have already been analyzed may be extended and overridden, thus possibly rendering the previous analysis information incorrect. Let us illustrate this issue with an example:

```
class A {
   void m(){/* code for A.m() */};
   void caller_m(){this.m();};
};
class B extends A {
   void m(){/* code for B.m() */};
};
```

```
class Example {
   void method_main(A a){
      a.caller_m();
   };
};
```

[1] These analysis results have been obtained ignoring possible exceptions thrown by the Java virtual machine (e.g., no method found, unable to create object, etc.) for clarity of the presentation.

Here, there are three different classes: `A`, `B`, and `Example`. But for now, let us concentrate on classes `A` and `Example` only. If `A` is analyzed, the result obtained for `caller_m` depends directly on the result obtained for `A.m` (for instance, `caller_m` could be guaranteed to terminate under the condition that `A.m` terminates). Then, the class `Example` is analyzed, using the analysis results obtained for `A`. Let us suppose that analysis concludes that `method_main` terminates.

Now, let us suppose that `B` is added to the program. As shown in the example, `B` extends `A` and overrides `m`. Imagine now that the analysis concludes that the new implementation of `m` is not guaranteed to terminate. The important point now is that the termination behavior of some of the methods we have already analyzed can be altered, and we have to make sure that analysis results can correctly handle this situation. In particular, `caller_m` is no longer guaranteed to terminate, and the same applies to `method_main`. Note, however, that class inheritance is correctly handled by the analyzer if all the code (in this case the code of `B`) is available from the beginning. This is done by considering, at the invocation program point, the information about both implementations of `m`. However, in general, the analyzer does not know, during the analysis of `A`, that the class will be extended by `B`. Such a situation is very common in the analysis of libraries, since they must be analyzed without knowing which user-defined classes will override their methods. In this example, corresponds to `A` and `Example` being library classes and `B` being defined by the user.

In order to avoid this kind of problems, the concept of *contract* can be used (in the sense of *subcontracting* of [20]). This means that the analysis result for a given method `m` is taken as the contract for m, i.e., information about how `m` and any redefinition of it is supposed to behave with respect to the analysis of interest. A *contract*, same as an assertion, has two parts: the calling *preconditions* which must hold in order the contract can be applicable; and the *postcondition*, the result of the analysis with respect to that preconditions. For example, a contract for `A.m()` may say that it terminates under the condition that the *this* object of type `A` is an acyclic data structure. In the example above, when `B` is added to the program, we have to analyze `B.m` taking as call pattern the precondition (Pre) in the contract for `A.m`. This guarantees that the result obtained for `B.m` will be valid in the same set of input states as the contract for `A.m`. Then, we need to compare the postconditions. If $m_B{:}Pre \mapsto Post^B$ and $m_A{:}Pre \mapsto Post^A$ are the assertions generated for `B.m` and `A.m`, respectively, and Pre is the default calling pattern for both implementations, there are two possible cases: (a) If $Post^B \sqsubseteq Post^A$ then `B.m` satisfies the contract for `A.m`; (b) otherwise, the contract cannot be applied, and `B.m` is considered incorrect. The user can manually inspect the code of `B.m` and if the analyzer loses precision, add an assumed assertion for `B.m`. Interfaces and abstract methods are similar to overriding methods of a superclass, with the difference that there is no code to analyze in order to generate the contract. In this case, assumed assertions written by the user can be used as contracts.

6 Experiments

After obtaining the call graph for the classes of phoneME's java.lang package, a bottom-up traversal of the call graphs has been performed. In a few particular cases, it was required to enable other analyses included in COSTA (e.g., field sensitive analysis [5], as mentioned above) for proving termination, or disabling some features such as handling jvm exceptions.

Table 1 shows the results of termination analysis of java.lang package, plus some other packages used by java.lang. This table compares the analysis using the modular analysis described in this paper with the non-modular analysis previously performed by COSTA. The columns under **Modular** show the modular analysis results, while under the **Non Modular** heading non-modular results are shown. **#B**$_c$ shows the number of bytecode instructions analyzed for all methods in the corresponding class, **#T** shows the number of methods of each class for which COSTA has proved termination and **Time**$_a$ shows the analysis time of all the methods in each class. In the modular case, the total analysis time is **Time**$_a$ plus **T**$_{cg}$, the time spent building the call graph of each class.

The two columns under **Assumed** show the number of methods for which assumed assertions were required: **Nat** is the number of native methods in each class, and **NNat** contains the number of non-native methods that could not be proved terminating. Finally, the last two columns under **Related** contain the

Table 1. Termination Analysis for java.lang package in COSTA (execution times are in seconds)

Class	**Modular**				**Non Modular**			**Assumed**		**Related**	
	#B$_c$	#T	T$_{cg}$	Time$_a$	#B$_c$	#T	Time$_a$	Nat	NNat	1st	All
Boolean	56	6	0.02	0.19	67	6	0.22	0	0	1	1
Byte	59	7	0.40	0.22	1545	7	21.10	0	0	4	22
Character	64	11	0.16	0.27	513	11	1.03	0	0	6	11
Class	110	4	1.17	1.10	4119	3	842.70	11	1	20	58
Double	107	17	3.66	1.12	107	13	0.36	2	0	8	57
Error	7	2	0.02	0.04	60	2	0.12	0	0	2	4
FDBigInt	1117	14	0.80	16.10	2513	12	158.39	0	2	23	47
Float	106	18	3.74	1.16	3105	15	5674.96	2	0	9	60
FloatingDecimal	3028	12	4.32	1201.10	3402	9	4983.88	0	8	49	64
Integer	469	21	1.35	18.76	4519	21	62.51	0	0	7	20
Long	268	11	0.64	10.99	2164	11	36.08	0	0	7	20
Math	207	16	0.14	0.67	212	16	0.69	6	0	3	3
NoClassDefFoundError	7	2	0.02	0.04	108	2	0.13	0	0	2	6
Object	737	3	0.21	46.21	891	3	129.31	5	0	7	28
OutOfMemoryError	7	2	0.02	0.03	170	2	0.18	0	0	2	8
Runtime	14	3	0.02	0.08	27	3	0.08	4	0	1	1
Short	59	7	0.39	0.24	1545	7	20.83	0	0	4	22
String	1784	39	5.88	21.11	8709	32	7217.43	6	3	34	120
StringBuffer	1509	37	6.74	11.01	14206	33	12103.35	0	0	37	86
System	45	7	0.38	0.31	2778	6	4864.33	5	0	11	62
Throwable	615	4	0.16	1.23	628	4	60.54	2	0	6	22
VirtualMachineError	7	2	0.02	0.04	108	2	0.14	0	0	2	6
Exception Classes (18)	136	38	0.61	0.74	3961	38	21.27	0	0	11	18
com/sun/* (7)	1584	26	5.55	22.36	11293	16	5161.29	0	0		
java/io/* (8)	106	11	1.47	0.65	2337	9	4983.35	0	0		
java/util/* (3)	265	13	0.88	3.33	2171	12	51.93	0	0		
Total	**12473**	**333**	**38.77**	**1359.10**	**71258**	**295**	**46396.17**	**43**	**14**	**256**	**746**

number of methods from other classes that are invoked by the methods in the class, either directly, shown in **1st** or the total number of methods transitively invoked, shown in **All**. Some rows in the table contain results accumulated for a number of classes (in parenthesis). The last three rows in the table contain accumulated information for methods directly or transitively invoked by the java.lang package which belong to phoneME packages other than java.lang.These rows do not include information about **Related** methods, since they are already taken into account in the corresponding columns for java.lang classes. The last row in the table, **Total**, shows the addition for all classes of all figures in each column. A number of interesting conclusions can be obtained from this table. Probably, the most relevant result is the large difference between the number of bytecode instructions which need to be analyzed in the modular and non-modular cases: 12,473 vs 71,258 instructions, i.e. nearly 7 times more code needs to be analyzed in the non-modular approach. The reason for this is that though in the modular approach methods are (at least in principle) analyzed just once, in the non-modular approach methods which are required for the analysis of different initial methods are analyzed multiple times. Obviously, this difference in code size to be analyzed has a great impact on the analysis times: the **Total** row shows that the modular analysis of all classes in java.lang is more than 30 times faster than the non-modular case.

Another crucial observation is that by using the modular approach we have been able to prove termination of 38 methods for which the non-modular approach is not able, either because the analysis runs out memory or because it fails to produce results within a reasonable time. Furthermore, the modular approach in this setting has turned out to be strictly more precise than the non-modular approach, since for all cases where the non-modular approach has proved termination, it has also been proved by the modular approach. This results in 333 methods for which termination has been proved in the modular approach, versus 295 in the non-modular approach. Altogether, in our experiments we have tried to prove termination of 389 methods. In the studied implementation of JavaME, 43 of those methods are native. Therefore, COSTA could not analyze them, and assumed assertions have been added for them. In addition, COSTA was not able to prove termination of 14 methods, neither in the modular nor non-modular approaches, as shown in the **NNat** column. For these methods, assumed assertions have also been added, and have not been taken into account in the other columns except in the last two ones. These two columns provide another view on the difference between using modular and non-modular analyses with respect to the number of transitively invoked methods (746) that required analysis, w.r.t. those directly invoked (256). In the modular case, only directly invoked methods need to be considered, and only for loading their assertions, whereas the non-modular approach requires loading (and analyzing) all related methods. We now describe in more detail the methods whose termination has not been proved by COSTA and the reasons for this:

– **Bitwise operations.** The size analysis currently available in COSTA is not capable of tracking numeric values after performing bitwise operations on them. Therefore, we cannot prove termination of some library methods which perform bitwise operations (in most cases, right or left shift operations) on variables which affect a loop termination condition.

– **Arrays issues.** During size analysis, arrays are abstracted to their size. Though this is sufficient for proving termination of many loops which traverse arrays, termination cannot be proved for loops whose termination depends on the value of specific elements in the array, since such values are lost by size abstraction.
– **Concurrency.** Though it is the subject of ongoing work, COSTA does not currently handle concurrent programs. Nonetheless, it can handle Java code in which `synchronized` constructs are used for preventing thread interferences and memory inconsistencies. In particular, few `java.lang` phoneME classes make real use of concurrency. For this reason, `Thread` class has not been included in the test, neither Table 1 does include information regarding `Class.initialize` nor `wait` methods defined in `Object`.
– **Unstructured control flow.** There are some library methods in which the control flow is unstructured, apparently for efficiency reasons. For example, `String.indexOf` uses a *continue* statement wrapping several nested loops, the outer most of them being an endless loop as in the following code (on the left):

```
indexOf(String str, int i){
 ...                              fixResourceName(String n){
 searchChar:                       int stI = 0;
  while (true) {                   int e = 0;
   ...
   if (i > max) return -1;         while((e=n.indexOf('/',stI))!= -1){
   while (j < end) {                if (e == stI) {
    if (v1[j++] != v2[k++]){         stI++;  continue;}
     i++; continue searchChar;}}       .... } } }
   return i - offset;} }
```

– **Other Cases.** `ResourceInputStream.fixResourceName` involves a call to a native method in the loop condition (see code above on the right). A termination assertion is not enough to find a ranking function of the loop to prove termination.

7 Discussion

Modular analysis has received considerable attention in different programming paradigms, ranging from, e.g., logic programming [14,11,8] to object-oriented programming [22,6,19]. A general theoretical framework for modular abstract interpretation analysis was defined in [13], but most of the existing works regarding modular analysis have focused on specific analyses with particular properties and using more or less ad-hoc techniques. A previous work from some of the authors of this paper presents and empirically tests a modular analysis framework for logic programs [14,11]. There are important differences with this paper: in addition to the programming paradigm, the framework of [14] is designed to handle one abstract domain, while the framework presented in this paper handles several domains at the same time, and the previous work is based on `CiaoPP`, a polyvariant context-sensitive analyzer in which an intermodular fixpoint algorithm was performed. In [22] a control-flow analysis-based technique is proposed for call graph construction in the context of OO languages. Although there have been other works in this area, the novelty of this approach is that it is context-sensitive. Also, [6] shows a way to perform modular class analysis by translating the OO program into *open* DATALOG programs. In [19] an abstract interpretation based approach to the analysis of class-based, OO languages is presented. The analysis is split in two separate semantic functions, one for the analysis of an object and another one for the analysis of the context that uses that object. The interdependence between context and object is expressed by two mutually recursive equations. In addition, it is context-sensitive and polyvariant. As conclusion, in this work we have presented an approach which is, to the best of our knowledge, the first modular termination analysis for OO languages. Our approach is based on the use of assertions as communication mechanism between the analysis of different methods. The experimental results show that the approach increases the applicability of termination analysis. The flexibility of this approach allows a higher level of scalability and makes it applicable to component-based systems, since is not required that all code be available to the analyzer. Furthermore, the specification obtained for a component can be reused for any other component that uses it. It remains as future work to extend the approach to other intermediate cases between modular and global analysis, i.e., by allowing analysis of several methods as one unit, even if they are not in the same cycle. This can be done without technical difficulties and it should be empirically determined what granularity level results in more efficient analysis.

Acknowledgments. The authors would like to thank Damiano Zanardini for interesting discussions and for his help with the heap analysis in COSTA. This work was funded in part by the Information & Communication Technologies program of the European Commission, Future and Emerging Technologies (FET), under the ICT-231620 *HATS* project, by the Spanish Ministry of Science and Innovation (MICINN) under the TIN-2008-05624 *DOVES* project, the TIN2008-04473-E (Acción Especial) project, the HI2008-0153 (Acción Integrada) project, the UCM-BSCH-GR58/08-910502 Research Group and by the Madrid Regional Government under the S2009TIC-1465 *PROMETIDOS* project.

References

1. Albert, E., Arenas, P., Codish, M., Genaim, S., Puebla, G., Zanardini, D.: Termination Analysis of Java Bytecode. In: Barthe, G., de Boer, F.S. (eds.) FMOODS 2008. LNCS, vol. 5051, pp. 2–18. Springer, Heidelberg (2008)
2. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Cost analysis of java bytecode. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 157–172. Springer, Heidelberg (2007)

3. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: COSTA: Design and Implementation of a Cost and Termination Analyzer for Java Bytecode. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2007. LNCS, vol. 5382, pp. 113–132. Springer, Heidelberg (2008)
4. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Resource Usage Analysis and its Application to Resource Certification. In: FOSAD 2007. LNCS, vol. 5705, pp. 258–288. Springer, Heidelberg (2009)
5. Albert, E., Arenas, P., Genaim, S., Puebla, G., Ramírez, D.: From Object Fields to Local Variables: A Practical Approach to Field-Sensitive Analysis. In: Cousot, R., Martel, M. (eds.) SAS 2010. LNCS, vol. 6337, pp. 100–116. Springer, Heidelberg (2010)
6. Besson, F., Jensen, T.: Modular class analysis with datalog. In: Cousot, R. (ed.) SAS 2003. LNCS, vol. 2694, pp. 19–36. Springer, Heidelberg (2003)
7. Bradley, A.R., Manna, Z., Sipma, H.B.: Termination of polynomial programs. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 113–129. Springer, Heidelberg (2005)
8. Codish, M., Debray, S.K., Giacobazzi, R.: Compositional analysis of modular logic programs. In: Proc. POPL 1993 (1993)
9. Codish, M., Taboch, C.: A semantic basis for the termination analysis of logic programs. J. Log. Program. 41(1), 103–123 (1999)
10. Cook, B., Podelski, A., Rybalchenko, A.: Termination proofs for systems code. In: PLDI (2006)
11. Correas, J., Puebla, G., Hermenegildo, M., Bueno, F.: Experiments in Context-Sensitive Analysis of Modular Programs. In: Hill, P.M. (ed.) LOPSTR 2005. LNCS, vol. 3901, pp. 163–178. Springer, Heidelberg (2006)
12. Cousot, P., Cousot, R.: Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: POPL 1977, pp. 238–252. ACM, New York (1977)
13. Cousot, P., Cousot, R.: Modular static program analysis. In: CC 2002. LNCS, vol. 2304, pp. 159–179. Springer, Heidelberg (2002)
14. Puebla, G., et al.: A Generic Framework for Context-Sensitive Analysis of Modular Programs. In: Bruynooghe, M., Lau, K. (eds.) Program Development in Computational Logic. LNCS, vol. 3049, pp. 233–260. Springer, Heidelberg (2004)
15. Genaim, S., Zanardini, D.: The acyclicity inference of COSTA. In: 11th International Workshop on Termination (July 2010)
16. Giesl, J., Schneider-Kamp, P., Thiemann, R.: AProVE 1.2: Automatic Termination Proofs in the Dependency Pair Framework. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 281–286. Springer, Heidelberg (2006)
17. Lee, C.S., Jones, N.D., Ben-Amram, A.M.: The size-change principle for program termination. In: POPL 2001, pp. 81–92. ACM, New York (2001)
18. Lindenstrauss, N., Sagiv, Y.: Automatic termination analysis of logic programs. In: ICLP (1997)
19. Logozzo, F.: Separate Compositional Analysis of Class-based Object-oriented Languages. In: Rattray, C., Maharaj, S., Shankland, C. (eds.) AMAST 2004. LNCS, vol. 3116, pp. 334–348. Springer, Heidelberg (2004)
20. Meyer, B.: Object-Oriented Software Construction, 2nd edn. Prentice-Hall, Inc., Upper Saddle River (1997)
21. Otto, C., Brockschmidt, M., von Essen, C., Giesl, J.: Termination Analysis of Java Bytecode by Term Rewriting. In: Waldmann, J. (ed.) WST 2009, Leipzig, Germany (June 2009)

22. Probst, C.W.: Modular Control Flow Analysis for Libraries. In: Hermenegildo, M.V., Puebla, G. (eds.) SAS 2002. LNCS, vol. 2477, pp. 165–179. Springer, Heidelberg (2002)
23. Spoto, F., Hill, P.M., Payet, E.: Path-length analysis of object-oriented programs. In: EAAI 2006. ENTCS. Elsevier, Amsterdam (2006)
24. Spoto, F., Jensen, T.: Class analyses as abstract interpretations of trace semantics. ACM Trans. Program. Lang. Syst. 25(5), 578–630 (2003)
25. Spoto, F., Mesnard, F., Payet, É.: A Termination Analyser for Java Bytecode based on Path-Length. ACM TOPLAS 32(3) (2010)

Decomposition of Constraint Automata

Bahman Pourvatan[1], Marjan Sirjani[2], Farhad Arbab[3], and Marcello M. Bonsangue[3]

[1] AmirKabir University, Tehran, Iran and LIACS, Leiden University, Leiden, Netherland
[2] Reykjavik University, Reykjavik, Iceland and University of Tehran, Tehran, Iran
[3] CWI, Amsterdam and LIACS, Leiden University, Leiden, Netherland

Abstract. Reo is a coordination language that can be used to model different systems. Constraint automata form a formal semantics for Reo connectors based on a co-algebraic model of streams. In this paper, we introduce complete constraint automata (*CCA*) whose extra information about entropy states helps in analyzing and decomposing them into Reo circuits. We show that a complete constraint automaton is invertible. This property helps to partition and decompose a constraint automaton, a process which can be utilized to synthesize Reo circuits from constraint automata, automatically.

Keywords: Reo, Constraint automata, Automata decomposition, Complete constraint automata, Inverse Automata.

1 Introduction

Reo [1] is an exogenous coordination language, wherein complex connectors are compositionally built out of simpler ones. The simplest connectors in Reo consist of a set of channels with well-defined behavior. Reo connectors are represented graphically as circuits that resemble electronic circuits. The emphasis in Reo is on the connectors, which orchestrate the synchronization and communication among components, not on the internal behavior of components. Constraint automata [2] were introduced as a compositional semantics for Reo. Using constraint automata we can analyze the behavior of Reo circuits. Although Reo was initially introduced as a coordination language, it can be used as a model of concurrency in various kinds of applications. Reo and constraint automata are used as an ADL (Architectural Description Language) [3], as a system-level design language in hardware-software co-designs [4], and as an orchestration language for web services [5]. In these applications, Reo is generally used to show the communication and synchronization, and constraint automata are used to model the components. In this way, the behavior of a whole system can be compositionally constructed using the constraint automata specification of its constituents.

In this paper, we propose an effective decomposition approach for constraint automata. Given a constraint automaton that specifies the desired behavior of a system, and a number of constraint automata, each describing the behavior of a component or sub-connector that can be used to construct the system, our decomposition technique produces the missing behavior necessary to compose those components (and/or sub-connectors) into the specified system. We provide an approach for automatic decomposition. As we derive the constituent components from the specified functionality by decomposition, it is guaranteed that the resulting composed system fulfills the requirement and no further verification or validation is necessary.

Our approach in this paper exploits the structure of a constraint automaton, instead of the composition of its SDS language. Our proposed approach can decompose a constraint automaton into two constraint automata, one of which is a given operand of the product of the two. The product on constraint automata has no inverse: for a constraint automaton $\mathcal{A}$ built as the product of two constraint automata $\mathcal{B}$ and $\mathcal{C}$ ($\mathcal{A} = \mathcal{B} \bowtie \mathcal{C}$) it is not possible to obtain $\mathcal{B}$, given the two constraint automata $\mathcal{A}$ and $\mathcal{C}$. The synthesis approach in [2] does not address this problem, either.

For such "anchored" decomposition of constraint automata, we need the inverse of the product operation. Instead of defining this inverse, we use the product operation itself with the inverse of constraint automata. To define the inverse of constraint automata, we define $Complete\ Constraint\ Automata\ (CCA)$ by adding extra information about the impossible events and states disallowed by the original constraint automata. Complete constraint automata are invertible and we define the $Inverse\ of\ Complete\ Constraint\ Automata$, formally.

Plan of the paper. The rest of this paper is organized as follows: Section 2 contains a brief overview of Reo, channels, and connectors. In Section 3 we briefly discuss constraint automata. In Section 4, we define complete constraint automata. In Section 5, we show an example for decomposition. Related work is discussed in Section 6. In Section 7, we discuss our future work and conclude the paper.

2 Reo: A Coordination Language

Reo is a formal language for building component connectors in a compositional manner [2]. Reo allows one to model the behavior of connectors, formally reason about them, and once proven correct, automatically generate the so-called glue code from such specification. Each connector in Reo is, in turn, constructed compositionally out of simpler connectors, which are ultimately composed out of a set of primitive connectors, called channels.

A channel is a primitive communication medium with exactly two ends, each with its own unique identity, plus a (set of) constraint(s) relating the flows of data through those ends. There are two types of channel ends: source end through which data enters, and sink end through which data leaves a channel. A channel must support a certain set of primitive operations, such as I/O, on its ends; beyond that, Reo places no restriction on the behavior of a channel. This allows an open-ended set of different channel types to be used simultaneously together in Reo, each with its own policy for synchronization, buffering, ordering, computation, data retention/loss, etc [2]. Reo offers a set of open ended channels, but there is also a set of primitive channels, shown in Figure 1, out of which the most useful circuits with typical coordination patterns can be built. We mostly use these channels in our examples.

A *Sync* channel has a source end and a sink end. The pair of I/O operations on its two ends can succeed only simultaneously, while the channel passes the data item on its source end to its sink end. A *LossySync* is similar to *Sync* but its source end always accepts data items. The data item at the source end is lost if there is no reader at the sink end. If the sink end is ready to accept then the channel transfers the data item exactly the same as a *Sync* channel. *SyncDrain* has two source ends. The pair of I/O operations on its two ends can succeed only simultaneously and all data items written to this channel are lost. A *FIFO1* has a source and a sink end and an internal buffer with the capacity for 1 data item. The channel accepts a data item at its source end only if its internal buffer is empty. The accepted data item is kept in the internal buffer and makes it full. An appropriate operation (take) on the sink end succeeds only if the buffer is not empty. A *Filter* has a source and a sink and a specified pattern P. An operation on the source end of this channel that attempts to writes a data item that does not match P, succeeds immediately and the channel loses the data item. Writing a data item that matches P to the source of this channel succeeds only simultaneously together with a take operation on its sink, which obtains the data item.

Channels are connected to make a circuit. Connecting channels is achieved by joining their channel ends together into *nodes*. Thus, a *node* consists of a set of channel ends. A node in Reo has a certain semantics which depends on its type. A node that contains only source channel ends is called a source node; one that contains only sink channel ends is called a sink node; and one that contains both types of channel ends is called a mixed node. The source and sink nodes of a connector are also collectively called its boundary nodes.

A component can write data items to a source node that it is connected to. The write operation succeeds only if all (source) channel ends coincident on the node accept the data item, in which case the data item is transparently written to every source end coincident on the node. A source node, thus, acts as a *replicator*. A component can obtain data items, by an input operation, from a sink node that it is connected to. A take operation succeeds only if at least one of the (sink) channel ends coincident on the node offers a suitable data item; if more than one coincident channel end offers suitable data items, one is selected nondeterministically. A sink node, thus, acts as a nondeterministic *merger*. A mixed node nondeterministically selects and takes a suitable data item offered by one of its coincident sink channel ends and replicates it into all of its coincident source channel ends.

The behavior of every connector in Reo imposes a specific coordination pattern on the entities that perform normal I/O operations on the boundary nodes of that connector, without the knowledge of those entities. This makes Reo a powerful glue language for compositional construction of connectors to combine component instances into a software system, and exogenously orchestrate their mutual interactions.

3 Constraint Automata: Compositional Semantics of Reo

Constraint automata are presented in [2] as formal semantics for Reo connectors based on a co-algebraic semantics given in [6]. Using constraint automata as an operational model for Reo connectors, the automata states stand for the possible configurations (e.g., the contents of the *FIFO*-channels of a Reo connector) while the automata-transitions represent the possible data flows and their effects on these configurations. The operational semantics for Reo presented in [2] can be reformulated in terms of constraint automata. The constraint automaton of a given Reo connector can be constructed by composition the constraint automata of its primitive channels. The constraint automata composition operators for carrying out this concatenation are presented in [2].

An extension of constraint automata with state memory(*CASM*) extends the language of data constraints to accommodate state memory cells.

Definition 1 **(*Constraint Automata with State Memory (*CASM*)*).** A constraint automaton with state memory (over the data domain $Data$) is a tuple $\mathcal{A} = (Q, \mathcal{N}, \rightarrow, q_0, \mathcal{M})$ where

- Q is a finite set of states
- $\mathcal{N}$ is a finite set of names.
- $\rightarrow$ is a finite subset of $Q \times 2^{\mathcal{N}} \times DC(\mathcal{N}, \mathcal{M}, Data) \times Q$, called the transition relation of $\mathcal{A}$, where $DC(\mathcal{N}, \mathcal{M}, Data)$ is the set of data constraints, defined below.
- $q_0 \in Q$ is an initial state.
- $\mathcal{M}$ is a set of memory cell names, where $\mathcal{N} \cap \mathcal{M} = \emptyset$. □

We can partition $\mathcal{N}$ into three disjoint sets, $\mathcal{N}^{src}$ a set of source node names, $\mathcal{N}^{snk}$ a set of sink node names, and $\mathcal{N}^{mix}$ a set of mixed node names. Thus $\mathcal{N} = \mathcal{N}^{src} \uplus \mathcal{N}^{snk} \uplus \mathcal{N}^{mix}$.

We write $q \xrightarrow{N,g} p$ instead of $(q, N, g, p) \in \rightarrow$. We call $N \in 2^{\mathcal{N}}$ the name-set and g the guard or data constraint of the transition. For every transition $q \xrightarrow{N,g} p$ we require that $g \in DC(N, \mathcal{M}, Data)$, where $DC(N, \mathcal{M}, Data)$ is the language defined by the following grammar:

$$g \ ::= \ true \ \mid \ \neg g \ \mid \ g \wedge g \ \mid \ u = u$$

$$u \ ::= \ d(n) \mid m' \mid m \mid v$$

Here "=" is the symmetric equality relation, $n \in N$ is a port name, $d(n)$ refers to the data item that passes through port n, $m \in \mathcal{M}$ refers to a memory cell in the current state (source of the transition), m' refers to the memory cell m in the next state (target of the transition), and $v \in Data$. As a shorthand, we allow in our syntax *false* to stand for $\neg$ *true*, and other logical operators, such as $\vee$ and $\implies$ (for implication), can also be defined, in the usual way.

We omit transitions whose data constraints can be reduced to $\neg$ *true* using the boolean laws. A data constraint, g, that can be reduced to *true* can be left out. We use $\mathcal{M}_g$ to denote the set of all $m \in \mathcal{M}$ that syntactically appear as m in a data constraint g; and $\mathcal{M}'_g$ to denote the set of all $m \in \mathcal{M}$ that syntactically appear as m' in g.

We use a valuation function $\mathcal{V}_q : \mathcal{M} \rightarrow 2^{Data}$ to designate the set of values $\mathcal{V}_q(m)$ of a memory cell $m \in \mathcal{M}$ in a state $q \in Q$, where $\mathcal{V}_{q_0}(m) = \emptyset$ for all $m \in \mathcal{M}$. A constraint automaton can make a transition $q \xrightarrow{N,g} p$ only if there exists a substitution for every syntactic element $d(n)$, m, and m' that appears in g to make it *true*. A substitution simultaneously replaces every occurrence of $d(n)$ with the data value (to be) exchanged through the node $n \in N$; every occurrence of m with a value $v \in \mathcal{V}_q(m)$; and every occurrence of m' with a value $v \in Data$. Making this transition, the automaton defines the valuation function $\mathcal{V}_p$ for the target state p, as follows. For every $m \in \mathcal{M}'_g$, $\mathcal{V}_p(m)$ is the set of all $v \in Data$ whose replacements in g yield substitutions that make g *true*. For every $m \in \mathcal{M} \setminus \mathcal{M}'_g$, $\mathcal{V}_p(m) = \emptyset$.

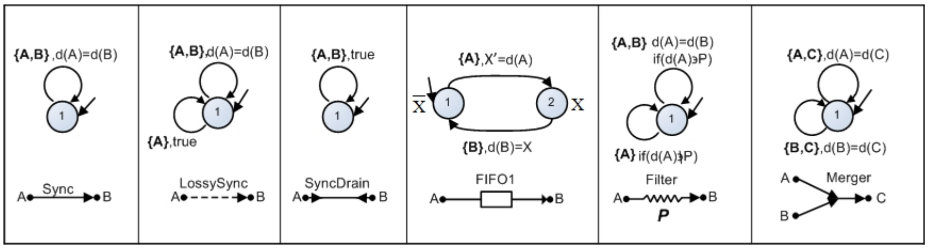


Fig. 1. Constraint automata with state descriptor for primitive channels and the merger

Figure 1 shows the constraint automata with state memory for the primitive channels and the merger node.

In order to find which nodes and/or memory cells are related by a data constraint g to each other or to values in $Data$, we use the function $(\_)^{\#}$ to parse g.

Definition 2 (Related Names of Data Constraints). *The related names of a data constraint g is the symmetric binary relation $g^{\#}$, derived from g by the following structural induction rules:*

- $(true)^{\#} = \emptyset$
- $(\neg g)^{\#} = g^{\#}$
- $(g_1 \wedge g_2)^{\#} = g_1^{\#} \cup g_2^{\#}$
- *for* $x \in \mathcal{N}, m \in \mathcal{M} : (d(x) = m)^{\#} = (d(x) = m')^{\#} = (m = d(x))^{\#} = (m' = d(x))^{\#} = \{(x, m), (m, x)\}$
- *for* $x \in \mathcal{N}, v \in Data : (d(x) = v)^{\#} = (v = d(x))^{\#} = \{(x, v), (v, x)\}$
- *for* $m \in \mathcal{M}, v \in Data : (m = v)^{\#} = (m' = v)^{\#} = (v = m)^{\#} = (v = m')^{\#} = \{(m, v), (v, m)\}$
- *for* $x, y \in \mathcal{N} : (d(x) = d(y))^{\#} = \{(x, y)\}$
- *for* $m, n \in \mathcal{M} : (m = n)^{\#} = (m' = n)^{\#} = (m = n')^{\#} = (m' = n')^{\#} = \{(m, n)\}$ □

3.1 Product of Constraint Automata with State Memory

The product of two constraint automata with state memory is defined in a similar way to the product of two constraint automata [2].

Definition 3. *[Product-automaton (join)]* The product-automaton of the two *CASM* $\mathcal{A}_1 = (Q_1, \mathcal{N}_1, \rightarrow_1, q_{0_1}, \mathcal{M}_1)$ and $\mathcal{A}_2 = (Q_2, \mathcal{N}_2, \rightarrow_2, q_{0_2}, \mathcal{M}_2)$ is:

$$\mathcal{A}_1 \bowtie \mathcal{A}_2 = (Q_1 \times Q_2, \mathcal{N}_1 \cup \mathcal{N}_2, \rightarrow, \langle q_{0_1}, q_{0_2} \rangle, \mathcal{M}_1 \cup \mathcal{M}_2)$$

where $\rightarrow$ is defined by the following rules:

$$\frac{q_1 \xrightarrow{N_1, g_1}_1 p_1, q_2 \xrightarrow{N_2, g_2}_2 p_2, N_1 \cap \mathcal{N}_2 = N_2 \cap \mathcal{N}_1}{\langle q_1, q_2 \rangle \xrightarrow{N_1 \cup N_2, g_1 \wedge g_2} \langle p_1, p_2 \rangle}$$

and

$$\frac{q_1 \xrightarrow{N,g}_1 p_1, N \cap \mathcal{N}_2 = \emptyset}{\langle q_1, q_2 \rangle \xrightarrow{N,g} \langle p_1, q_2 \rangle}$$

and latter's symmetric rule. □

The node sets $\mathcal{N}^{mix}$, $\mathcal{N}^{src}$, $\mathcal{N}^{snk}$ of the product are:

$$\begin{aligned}
\mathcal{N}^{src} &= \mathcal{N}_1^{src} \setminus (\mathcal{N}_2^{mix} \cup \mathcal{N}_2^{snk}) \cup \mathcal{N}_2^{src} \setminus (\mathcal{N}_1^{mix} \cup \mathcal{N}_1^{snk}) \\
\mathcal{N}^{snk} &= \mathcal{N}_1^{snk} \setminus (\mathcal{N}_2^{mix} \cup \mathcal{N}_2^{src}) \cup \mathcal{N}_2^{snk} \setminus (\mathcal{N}_1^{mix} \cup \mathcal{N}_1^{src}) \\
\mathcal{N}^{mix} &= \mathcal{N}_1^{mix} \cup \mathcal{N}_2^{mix} \cup (\mathcal{N}_1^{src} \cap \mathcal{N}_2^{snk}) \cup (\mathcal{N}_1^{snk} \cap \mathcal{N}_2^{src})
\end{aligned}$$

Moving toward Complete Constraint Automata. Complete constraint automata are based on *CASM*. First, we add a descriptor for each state, which shows whether each memory cell in that state has a value or is empty. Then, we define complete constraint automata and their inverse.

We define the descriptor of a state using the descriptor for a transition, based on the data constraints g. The function δ returns a set of memory cells that are referenced in the data constraint g of a transition in the form of m or $\overline{m}$. We define the function $\delta(q \xrightarrow{N,g} p)$ as:

$$\delta(q \xrightarrow{N,g} p) = \{m \mid m \in \mathcal{M} \wedge \mathcal{V}_p(m) \neq \emptyset\} \cup \{\overline{m} \mid m \in \mathcal{M} \wedge \mathcal{V}_p(m) = \emptyset\}$$

Observe that, by the definition of the valuation function $\mathcal{V}$ and how it affects the memory cells of p, a given $m \in \mathcal{M}$ can appear in $\delta(q \xrightarrow{N,g} p)$ exclusively either as m or as $\overline{m}$, depending on whether or not m' appears in a term in g. All descriptors have the same size ($\mid \delta(q \xrightarrow{N,g} p) \mid = \mid \mathcal{M} \mid$).

We consider a subset of constraint automata with state memory as constraint automata with state descriptor ($CASD$) defined as follows:

Definition 4. ***[Constraint Automata with State Descriptor (*CASD*)]*** *A constraint automaton with state memory $\mathcal{A} = (Q, \mathcal{N}, \rightarrow, q_0, \mathcal{M})$ is a constraint automaton with state descriptor if (1) for every $q \in Q$ all incoming transitions $p \xrightarrow{N,g} q$ have the same descriptor; and (2) no transition into the initial state imposes a constraint on its memory cells; i.e.:*

$$(p \xrightarrow{N_1,g_1} q) \wedge (r \xrightarrow{N_2,g_2} q) \Leftrightarrow \delta(p \xrightarrow{N_1,g_1} q) = \delta(r \xrightarrow{N_2,g_2} q)$$

$$\delta(q \xrightarrow{N,g} q_0) = \{\overline{m} \mid m \in \mathcal{M}\}$$

□

Definition 5. ***[Descriptor of a state]*** *For a constraint automaton with state descriptor, we extend the descriptor function δ to states. For each state q: $\delta(q) = \delta(p \xrightarrow{N,g} q)$, and $\delta(q_0) = \{\overline{m} \mid m \in \mathcal{M}\}$.* □

Every constraint automaton $\mathcal{A}$ can be mapped to a *CASD* $\mathcal{A}'$ by splitting each state of $\mathcal{A}$ with different incoming transition descriptors into multiple states in $\mathcal{A}'$ such that each state in $\mathcal{A}'$ has the same descriptor for all of its incoming transitions; and $\mathcal{A} \simeq \mathcal{A}'$(where $\simeq$ denotes bisimulation). Thus, the set of *CASD* covers *CA*.

Next, we define a subset of constraint automata with state descriptors as constraint automata with unique state descriptor, where the descriptor of every state is unique, i.e., every state has a descriptor that is different than the descriptors of all other states in the automaton.

Definition 6. ***[Constraint Automata with Unique State Descriptor (*CAUSD*)]*** *A constraint automaton with state descriptor $\mathcal{A} = (Q,\ \mathcal{N},\ \rightarrow,\ q_0,\ \mathcal{M})$ is a constraint automaton with unique state descriptor if for all p, $q \in Q$:*

$$\delta(p) = \delta(q) \Leftrightarrow p = q$$

□

Every *CASD* $\mathcal{A} = (Q,\ \mathcal{N},\ \rightarrow,\ q_0,\ \mathcal{M}_{\mathcal{A}})$ can be mapped to a bisimilar *CAUSD* $\mathcal{A}' = (Q,\ \mathcal{N},\ \rightarrow,\ q_0,\ \mathcal{M}_{\mathcal{A}'})$ by adding memory cells to distinguish the non-bisimilar states of $\mathcal{A}$ that have the same state descriptors. If $q_1, q_2 \in Q$, q_1 and q_2 are not bisimilar, and $\delta_{\mathcal{A}}(q_1) = \delta_{\mathcal{A}}(q_2)$, we add a new memory cell $m \notin \mathcal{M}_{\mathcal{A}}$ to $\mathcal{M}_{\mathcal{A}'}$ and define $\delta_{\mathcal{A}'}(q_1) = \delta_{\mathcal{A}}(q_1) \cup \{m\}$ and $\delta_{\mathcal{A}'}(q_2) = \delta_{\mathcal{A}}(q_2) \cup \{\overline{m}\}$. Thus, the set of *CAUSD* covers *CASD* (which, in turn, covers *CA*). According to Definition 6 a constraint automaton with unique state descriptor with $\mathcal{M} = \emptyset$ has only one state ($\mid Q \mid = 1$) with descriptor $\emptyset$. Figure 1 shows the *CAUSD* for some Reo channels.

The state descriptors of the *FIFO1* in Figure 1 are:

$$\delta(1) = \{\overline{X}\}\ ,\ \delta(2) = \{X\}\ \ \textit{where}\ \ \mathcal{M} = \{X\}\ \ \textit{and}\ \ Q = \{1, 2\}$$

The product of two *CAUSD* is bisimilar to a *CAUSD* because we only consider pairs of states with unique descriptor. The state descriptor for each state in the result of the product is the union of the descriptors of the product operands, i.e., for $\langle q_1, q_2 \rangle \in Q \Rightarrow \delta(\langle q_1, q_2 \rangle) = \delta(q_1) \cup \delta(q_2)$.

For each constraint automaton $\mathcal{A}$, we define its negation $\mathcal{A}^{\neg}$ as follows:

Definition 7. ***[Negation of Constraint Automata with Unique State Descriptor]*** *The negation of a constraint automaton $\mathcal{A} = (Q,\ \mathcal{N},\ \rightarrow,\ q_0,\ \mathcal{M})$ denoted as $\mathcal{A}^{\neg}$ is defined as $\mathcal{A}^{\neg} = (Q,\ \mathcal{N},\ \rightarrow_{\neg},\ q_0,\ \mathcal{M})$ where the transition relation $\rightarrow_{\neg}$ is obtained from $\rightarrow$ by negating its data constraints:*

$$\frac{p \xrightarrow{N,g} q \in \rightarrow}{p \xrightarrow{N,\neg g} q \in \rightarrow_{\neg}}$$

□

4 Complete Constraint Automata

In order to decompose a *CAUSD* we need a new operation that acts as the inverse of the product operation. Instead of defining the inverse of the product operation we use the product operation itself together with the inverse of a *CAUSD*. A *CAUSD* $\mathcal{A}$ corresponds to (i.e, recognizes or produces) a formal language $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{N}^*$ and has an inverse

CAUSD $\mathcal{B}$ where (1) $\mathcal{B}$ corresponds to the formal language $\mathcal{N}^* \setminus \mathcal{L}(\mathcal{A})$, and (2) the product of $\mathcal{A}$ and $\mathcal{B}$ is an automaton with a single state that recognize only the null language. To arrive at an algebraic system, for every $\mathcal{A}$ we need to have a unique $\mathcal{B}$ that satisfies the second condition. However, *CAUSD* do not have this property: for a *CAUSD*, we can find more than one *CAUSD* that fulfils the second condition. For example, for a *CAUSD* with a single state p and two transitions $p \xrightarrow{\{\overline{A},\overline{B}\},true} p$ and $p \xrightarrow{\{A,B\},true} p$, we can find three different *CAUSD* $\mathcal{A}_1$, $\mathcal{A}_2$, $\mathcal{A}_3$ that satisfy the second condition. Each of these three automata has a single state p and their transition sets are $\mathcal{A}_1 : \{p \xrightarrow{\{A,\overline{B}\},true} p\}$, $\mathcal{A}_2 : \{p \xrightarrow{\{\overline{A},B\},true} p\}$, and $\mathcal{A}_3 : \{p \xrightarrow{\{A,\overline{B}\},true} p\ ,\ p \xrightarrow{\{\overline{A},B\},true} p\}$. We introduce complete constraint automata (*CCA*) to solve this problem and make constraint automata invertible.

For every *CAUSD* $\mathcal{A}$ we define a complete constraint automaton $\mathcal{A}'$ as its counterpart. To construct the complete constraint automaton $\mathcal{A}'$ for a *CAUSD* $\mathcal{A}$, for each state $q \in \mathcal{A}$ we define $\mathcal{A}'$ to have both q and its complement state q^c, all transitions in $\mathcal{A}$, and also all transitions that are impossible in $\mathcal{A}$.

Unmasking Constraint Automata. The unmasked version of a *CAUSD* explicitly shows the absence of node names on its transitions and the empty (stuttering) transitions that cause the automaton to remain in each state.

Definition 8. *[Unmasked Constraint Automata] The unmasked version of a* CAUSD $\mathcal{A} = (Q, \mathcal{N}, \rightarrow_{\mathcal{A}}, q_0, \mathcal{M})$ *is the automaton* $\mathcal{A}^U = (Q, \mathcal{N}, \rightarrow, q_0, \mathcal{M})$ *where* $\rightarrow = \rightarrow_S \cup \rightarrow_{\mathcal{A}}$ *and* $\rightarrow_S$ *is the smallest relation such that* $\forall q \in Q : q \xrightarrow{\emptyset,true} q \in \rightarrow_S$. □

Especially for the unmasked constraint automata, we often use the syntax $p \xrightarrow{N \cup \overline{N},g} q$ where $\overline{N} = \{\overline{n} \mid n \in \mathcal{N} \setminus N\}$ for $(p, N, g, q) \in \rightarrow$.

For example in Figure 1, there is only one transition for the automaton of the *Sync* channel, when A and B fire simultaneously. If A and B do not fire together, it is implied that the automaton remains in state 1. This behavior is expressed by the implicit stuttering transition $1 \xrightarrow{\emptyset,true} 1$ (not shown) in Figure 1. In Figure 2-i the unmasked version of the automaton for the *Sync* channel shows this stuttering behavior explicitly as the transition $1 \xrightarrow{\overline{AB},true} 1$.

There is a unique unmask constraint automaton $\xi_{\mathcal{N}}$, called the null automaton, for a name set $\mathcal{N}$ defined as follows. The null automaton serves as the zero element with product and negation.

Definition 9. *[Null Automaton] The null automaton, for a name set* $\mathcal{N}$, *is* $\xi_{\mathcal{N}} = (\{q_0\}, \mathcal{N}, \emptyset, q_0, \emptyset)$. □

Since all non-initial states in the product $\mathcal{A} \bowtie \mathcal{A}^{\neg}$ become unreachable from its initial state, we have that $\mathcal{A} \bowtie \mathcal{A}^{\neg} \simeq \mathcal{A}^{\neg} \bowtie \mathcal{A} \simeq \xi_{\mathcal{N}_{\mathcal{A}}}$, and that $\mathcal{A} \bowtie \xi_{\mathcal{N}_{\mathcal{A}}} \simeq \xi_{\mathcal{N}_{\mathcal{A}}} \bowtie \mathcal{A} \simeq \xi_{\mathcal{N}_{\mathcal{A}}}$.

In the rest of this paper, we use the term constraint automata instead of unmasked *CAUSD* when the context makes the intention clear.

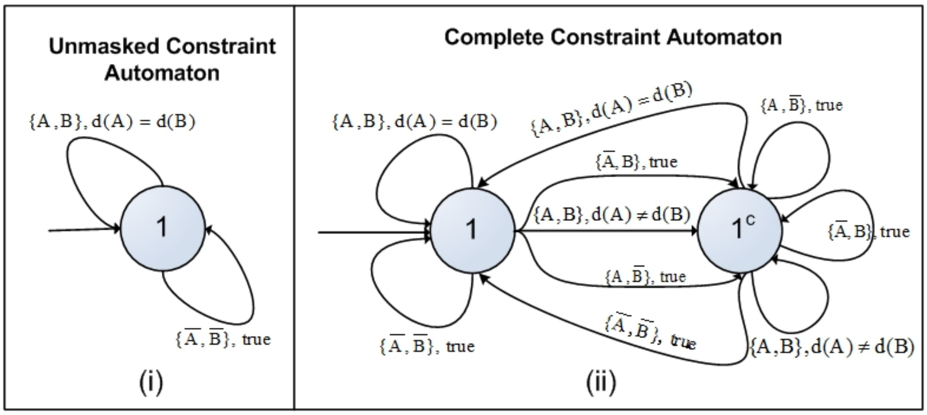


Fig. 2. Unmasked, and complete constraint automaton for *Sync* channel constraint automaton

Complete Constraint Automata. Constraint automata show the permissible transition in reaction to possible events, and unmasked constraint automata make the stuttering transitions explicit. In order to have an algebraic system over automata, we need to consider all impossible transitions as well. An automaton (*CAUSD*) with the set of names $\mathcal{N}$ may have any element of $2^{2^{\mathcal{N}}}$ on the outgoing transitions of each state. For each state $p \in Q$, the set $2^{2^{\mathcal{N}}} \setminus \ell_p$ contains all impossible combinations of names for the outgoing transitions of p, where $\ell_p = \{N \mid p \xrightarrow{N,g} q, \ \ q \in \ Q\}$. To construct the *CCA* from an automaton (*CAUSD*) for each p, we add the following impossible transitions: transitions with the elements of ℓ_p as their name sets with the negation of their corresponding data constraints, and the transitions with name sets of $2^{2^{\mathcal{N}}} \setminus \ell_p$ and true as their data constraints. For each state in the *CAUSD* automaton, we add a complement state as its entropy state in its *CCA* counterpart to which impossible transitions lead.

Figure 2-ii shows a complete constraint automaton for a *Sync* channel with the name set $\{A, B\}$. For the *Sync* channel, either A and B fire simultaneously and $d(A) = d(B)$ is true, or they do not fire at all. To build the complete constraint automaton of the *Sync* channel, we add the complement states, here $\{1^c\}$. For state 1, $\ell_1 = \{\{A, B\}, \{\overline{A}, \overline{B}\}\}$ and the elements of $2^{2^{\{A,B\}}} \setminus \ell_1$ are $\{\{A, \overline{B}\}, \{\overline{A}, B\}\}$. We add transitions to the complement state 1^c with $\{A, B\}$ and $\{\overline{A}, \overline{B}\}$ and the negated data constraints; and $\{\{A, \overline{B}\}, \{\overline{A}, B\}\}$ with the data constraints true. In Figure 2-ii, the transition with the name set $\{A, B\}$ with the $d(A) \neq d(B)$ data constraint is shown. The transition with the name set $\{\overline{A}, \overline{B}\}$ has a data constraint of false, and is, thus, removed.

We also add transitions from state 1^c back to 1 with the same labels as elements of ℓ_1 and we add the transitions from 1^c to 1^c for each transition from 1 to 1^c.

Definition 10 (***Complete Constraint Automata*** **(**CCA**)**). *A complete constraint automaton (*CCA*) (over the data domain Data) is a tuple $\mathcal{A} = (Q, \mathcal{N}, \rightarrow, q_0, \mathcal{M})$ where*

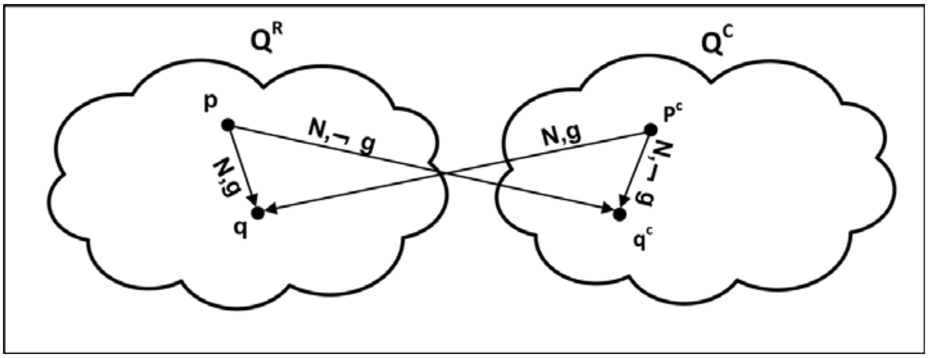


Fig. 3. Transition $p \xrightarrow{N,g} q$ and three added transitions in *CCA*

- *Q is a finite set of states that is partitioned into the Q^R and Q^C, and comes equipped with an isomorphism $(-)^C : Q^R \to Q^C$, such that the descriptor $\delta(q)$ of every state $q \in Q^R$ is unique among all states in Q^R, and $\delta(q) = \delta(q^c)$.*
- *$\mathcal{N}$ is a finite set of names.*
- *$\to$ is a finite subset of $Q \times 2^{\mathcal{N}} \times DC(\mathcal{N}, \mathcal{M}, Data) \times Q$, called the transition relation of $\mathcal{A}$, that satisfies the following rules, where $DC(\mathcal{N}, \mathcal{M}, Data)$ is the set of data constraints, as before.*
  - *For $p, q \in Q^R$ (as shown in Figure 3):*

$$\frac{p \xrightarrow{N,g} q}{p^c \xrightarrow{N,g} q \quad , \quad p \xrightarrow{N,\neg g} q^c \quad , \quad p^c \xrightarrow{N,\neg g} q^c}$$

  - *For $p \in Q^R$ and $\ell_p = \{N \mid p \xrightarrow{N,g} q,\ q \in Q^R\}$:*

$$\frac{N \in 2^{2^{\mathcal{N}}} \setminus \ell_p}{p \xrightarrow{N,true} p^c \quad , \quad p^c \xrightarrow{N,true} p^c}$$

$$\frac{(p \xrightarrow{N,g} q) \wedge (p \neq q)}{p \xrightarrow{N,true} p^c \quad , \quad p^c \xrightarrow{N,true} p^c}$$

- *q_0 is an initial state and $q_0 \in Q^R$.*
- *$\mathcal{M}$ is a finite set of memory cells.* □

Observe that for $p, q \in Q^R$, we have

$$\delta(p) = \delta(q) \Leftrightarrow p \simeq q \quad and \quad \delta(q) = \delta(q^c).$$

We can map any constraint automaton with unique state descriptor into its corresponding complete constraint automaton by applying the completion function $\Upsilon : CAUSD \to CCA$, defined below.

Definition 11 (***Completion of Constraint Automata with unique state descriptor***). *The completion of a constraint automaton with unique state descriptor $\mathcal{A} = (Q^R, \mathcal{N}, \to^R, q_0, \mathcal{M})$, denoted as $\Upsilon(\mathcal{A})$, is a constraint automaton $\mathcal{A}'$:*

$$\Upsilon(\mathcal{A}) = \mathcal{A}' = \ (Q^R \cup Q^C,\ \mathcal{N}, \rightarrow,\ q_0,\ \mathcal{M})$$

where

- $Q^C = \{q^c \mid q \in Q^R\}$.
- $\rightarrow$ *is defined by the following rules:*
  - *For each* $p,\ q \in Q^R$*:*

$$\frac{p \xrightarrow{N,g} q \in \rightarrow^R}{p^c \xrightarrow{N,g} q \ \ , \ \ p \ \xrightarrow{N,\neg g} \ q^c \ \ , \ \ p^c \ \xrightarrow{N,\neg g} \ q^c}$$

  - *For each* $p \in Q^R$ *and* ℓ_p=$\{N \mid p \xrightarrow{N,g} \ q, \ \ q \in \ Q^R\}$

$$\frac{N \in 2^{2^{\mathcal{N}}} \setminus \ \ell_p}{p \ \xrightarrow{N,true} \ p^c \ \ , \ \ p^c \ \xrightarrow{N,true} \ p^c} \qquad \frac{(p \xrightarrow{N,g} q) \wedge (p \neq q)}{p \ \xrightarrow{N,true} \ p^c \ \ , \ \ p^c \ \xrightarrow{N,true} \ p^c}$$

□

Observe that $\Upsilon(\mathcal{A})$ is also a complete constraint automaton. Figure 4 shows a complete constraint automaton for a *FIFO1* channel.

The state set in Q of every complete constraint automata on $\mathcal{A}$ can be split into two disjoint sets Q^R and Q^C, such that, (1) Q^R contains the initial state q_0; and (2) for every state p in Q^C, there exists a unique state q in Q^R with the same state descriptor as p with

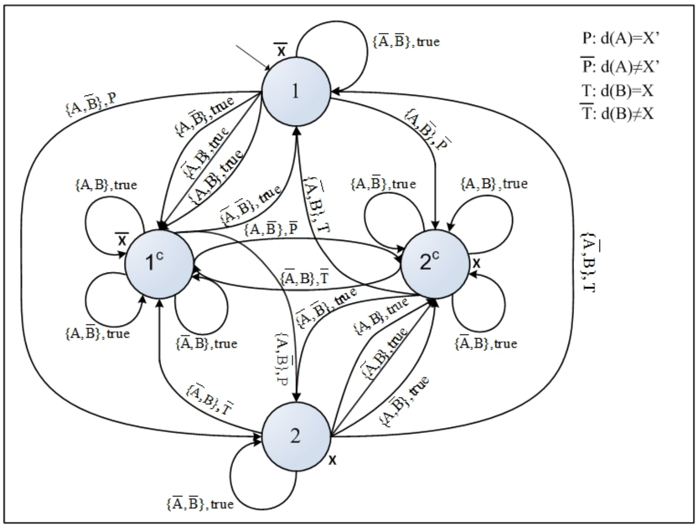


Fig. 4. Complete Constraint Automaton for *FIFO1* Channel

$\delta(q) = \delta(p)$, (3) there exists a τ transition from q to p, or p to q. By removing every transition $q \xrightarrow{N,g} p$ where p and q are not both in the same subset Q^R or Q^C, we obtain two disjoint constraint automata $\mathcal{A}_1 = (Q^R, \mathcal{N}, \rightarrow^R, q_0, \mathcal{M})$ and $\mathcal{A}_2 = (Q^C, \mathcal{N}, \rightarrow^C, q_0^c, \mathcal{M})$, such that, $\mathcal{A}_1 \bowtie \mathcal{A}_2 = \xi_{\mathcal{N}}$ and $\Upsilon(\mathcal{A}_1) = \mathcal{A}$.

Lemma 1. *The function Υ is an isomorphism.* □

4.1 Inverse of Complete Constraint Automata

Complete constraint automata are invertible. We define the inverse of complete constraint automata as follows:

Definition 12. ***[Inverse of Complete Constraint Automata]*** *The inverse of a complete constraint automaton $\mathcal{A} = (Q, \mathcal{N}, \rightarrow, q_0, M)$ is the complete constraint automaton $\overline{A} = (Q, \mathcal{N}, \rightarrow, q_0^c, M)$ where q_0 and q_0^c are related by the transition $q_0^c \xrightarrow{\emptyset, true} q_0$ or $q_0 \xrightarrow{\emptyset, true} q_0^c$, and $(-)^C$ is the isomorphism on Q in Definition 10.* □

Lemma 2. *For constraint automaton $\mathcal{A}$, we have $\Upsilon(\mathcal{A}^{\neg}) = \overline{\Upsilon(\mathcal{A})}$.* □

Figure 5, shows the inverse of complete constraint automata for the *Sync* and the *FIFO1* channels.

Lemma 3. *For complete constraint automaton $\mathcal{A}$ we have: $\mathcal{A} = \overline{\overline{\mathcal{A}}}$.* □

4.2 Product of Two Complete Constraint Automata

Complete constraint automata contain more structural information than *CASM*. Specifically, the isomorphism defined on the states set of a complete constraint automaton (in Definition 10) is a refinement that makes the normal product of *CASM* in Definition 3 inadequate for the *CCA*. Thus, we need a new definition for the $\bowtie$ product of the *CCA*. To do this, we first introduce the simple product of *CCA*, denoted as $\bigodot$, in Definition 13, as a straight-forward adaptation of Definition 3, which accommodates the extra information content of the *CCA* by classifying the states of the product automaton either as real or complement states.

Recall that we define complete constraint automata in order to use its inverse and the product operation, instead of defining the inverse of the product operation. For the complete constraint automata $A_1 = B \bowtie C$ and $A_2 = D \bowtie \overline{C}$, we expect $A_1 \bowtie A_2$ to be the same as $B \bowtie D$ (up to bisimilarity). But the simple product $A_1 \bigodot A_2$ yields the null automaton, instead of $B \bigodot D$. Therefore, the $\bigodot$ product cannot serve as the $\bowtie$ product of the *CCA*, and we need to consider more details to define the $\bowtie$ product on the *CCA*.

That the above example yields the null automaton follows from the fact that C and $\bar{C}$ are "contained" in A_1 and A_2, respectively. The $\bigodot$ product "fails" in such cases (by producing the null automaton, instead of the "expected" product), and these are precisely the cases that require special attention in the definition of the $\bowtie$ product. Intuitively, the $\bowtie$ product of the *CCA* identifies the part of an automaton whose inverse

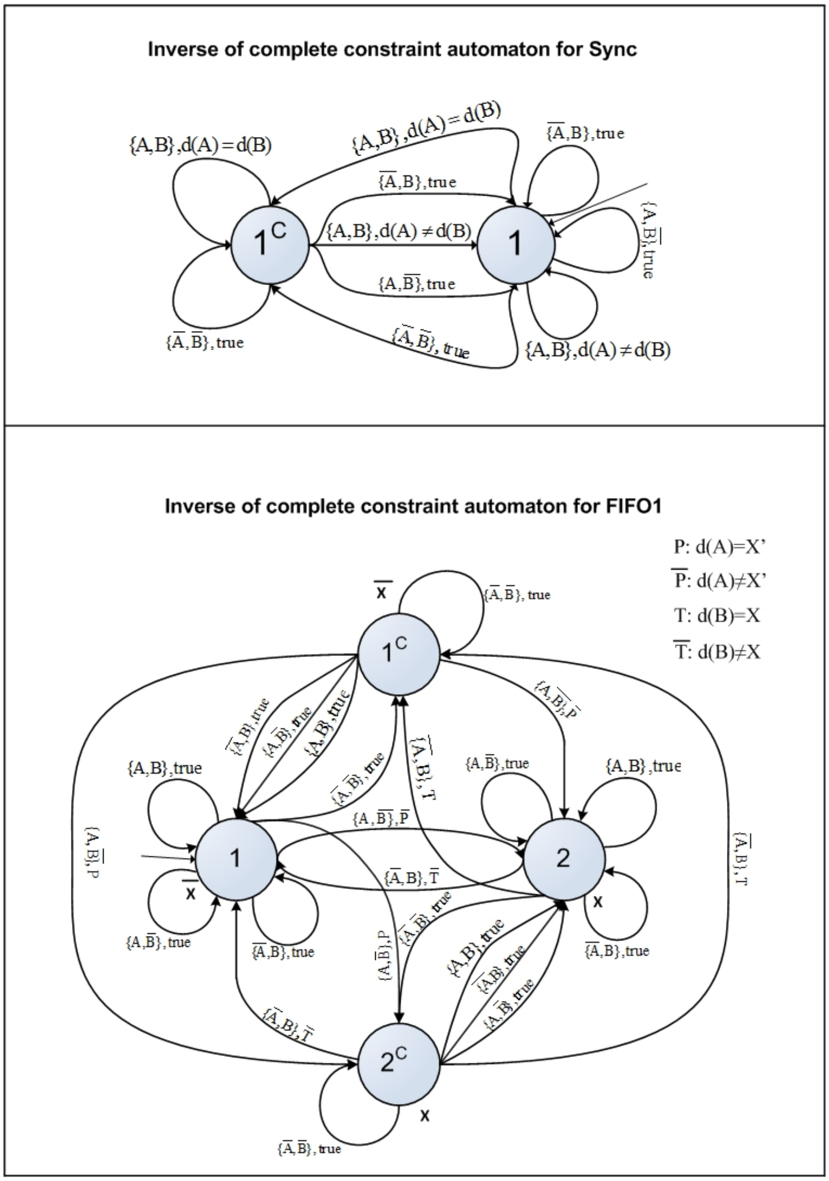

Inverse of complete constraint automaton for Sync
1^C
1
$\{A,B\}, d(A)=d(B)$
$\{A,B\}, d(A)=d(B)$
$\{\overline{A},B\}$, true
$\{\overline{A},B\}$, true
$\{A,B\}, d(A)\neq d(B)$
$\{A,\overline{B}\}$, true
$\{A,\overline{B}\}$, true
$\{\overline{A},\overline{B}\}$, true
$\{\overline{A},\overline{B}\}$, true
$\{A,B\}, d(A)\neq d(B)$
Inverse of complete constraint automaton for FIFO1
P: d(A)=X'
$\overline{P}$: d(A)≠X'
T: d(B)=X
$\overline{T}$: d(B)≠X
1^C
1
2
2^C
$\overline{X}$
$\overline{X}$
X
X
$\{\overline{A},\overline{B}\}$, true
$\{\overline{A},B\}$, true
$\{A,\overline{B}\}$, true
$\{A,B\}$, true
$\{\overline{A},\overline{B}\}$, true
$\{\overline{A},B\}$, T
$\{A,\overline{B}\}$, P
$\{A,B\}$, true
$\{A,\overline{B}\}$, true
$\{A,B\}$, true
$\{\overline{A},B\}$, true
$\{A,\overline{B}\}, \overline{P}$
$\{\overline{A},B\}, \overline{T}$
$\{A,\overline{B}\}, P$
$\{\overline{A},\overline{B}\}$, true
$\{A,B\}$, true
$\{\overline{A},\overline{B}\}$, true
$\{A,\overline{B}\}$, true
$\{\overline{A},B\}, \overline{T}$
$\{A,\overline{B}\}, P$
$\{\overline{A},B\}$, T
$\{\overline{A},\overline{B}\}$, true

Fig. 5. Inverse of the complete constraint automata for a *Sync* and a *FIFO1* channel

is contained in the other automaton (e.g., the automaton C and its inverse $\overline{C}$, above), subtracts (i.e., removes) this common overlap from both operands, and then forms the product of the remainders. In the rest of this section, we define the simple product, subtraction, and the product of the *CCA*.

Definition 13 (Simple Product of *CCA*). *The simple product of the two complete constraint automata* $\mathcal{A}_1 = (Q_1^R \cup Q_1^C, \mathcal{N}_1, \rightarrow_1, q_0^1, \mathcal{M}_1)$ *and* $\mathcal{A}_2 = (Q_2^R \cup Q_2^C, \mathcal{N}_2, \rightarrow_2, q_0^2, \mathcal{M}_2)$*, denoted as* $\mathcal{A}_1 \odot \mathcal{A}_2$ *is the automaton:*

$$\mathcal{A} = (Q^R \cup Q^C, \mathcal{N}_1 \cup \mathcal{N}_2, \rightarrow, \langle q_0^1, q_0^2 \rangle, \mathcal{M}_1 \cup \mathcal{M}_2)$$

where $\rightarrow$ *is defined by the rule:*

$$\frac{q_1 \xrightarrow{N_1, g_1}_1 p_1, q_2 \xrightarrow{N_2, g_2}_2 p_2, N_1 \cap \mathcal{N}_2 = N_2 \cap \mathcal{N}_1}{\langle q_1, q_2 \rangle \xrightarrow{N_1 \cup N_2, g_1 \wedge g_2} \langle p_1, p_2 \rangle}$$

and Q^R *and* Q^C *are subsets of* $(Q_1^R \cup Q_1^C) \times (Q_2^R \cup Q_2^C)$ *defined by the following rules:*

$$\frac{p \in Q_1^R, q \in Q_2^R}{\langle p, q \rangle \in Q^R} \qquad \frac{p \in Q_1^C \vee q \in Q_2^C}{\langle p, q \rangle \in Q^C}$$

□

The node sets $\mathcal{N}^{mix}$, $\mathcal{N}^{src}$, $\mathcal{N}^{snk}$ of the product are as in Section 3.

Note that the states with the same state descriptor in Q^C are bisimilar.

Figures 6 shows two complete constraint automata for two *Sync* channels and their product before and after merging the bisimilar complement states.

Subtracting a set of names $\mathcal{N}_1$ from an automaton $\mathcal{A}$ is a projection that yields another automaton $\mathcal{B}$. The name set of $\mathcal{B}$ is obtained from the name set of $\mathcal{A}$ by considering to remove all names in $\mathcal{N}_1$. A name in $\mathcal{N}_1$ can be removed only if is not related to any name not in $\mathcal{N}_1$. If a mixed node in $\mathcal{N}_1$ is not removed, its type may change to become a boundary node in $\mathcal{B}$.

Subtraction uses an asymmetric relation called the *directional data constraints* of a transition, which we use to derive potential paths for data-flows. As such, data constraints play a key role in our decomposition scheme. When no specific constraint needs to hold on the data exchanged through a node x in a transition, $d(x)$ simply does not appear in the data constraints of that transition. For instance, in Figure 1, the data constraint for the transition of the *CA* for the *SyncDrain* channel is *true*, which means any data item occurring on nodes A and B are acceptable. As a semantic model for Reo, a *CA* (or *CCA*) must reflect the characteristic property of Reo mixed nodes: they never produce, consume, or store data items; therefore, they cannot be the initial sources or the ultimate targets of any data transfer in any transition. The absence of data constraints, mentioned above, may lead to *CA* with transitions whose data constraints *seem to* indicate a mixed node is the initial source or the ultimate target of data transfers. To resolve this discrepancy, the implied meaning of the *absence* of any data constraint for a node must be made explicit in directional data constraints.

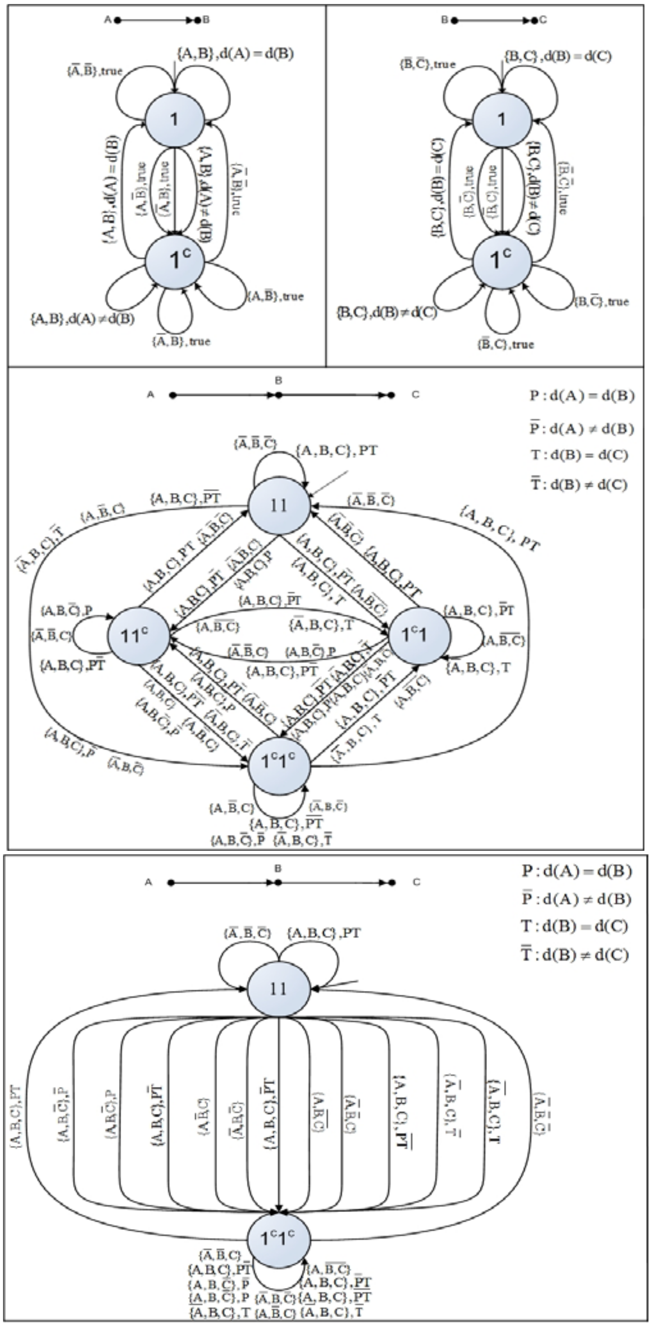


Fig. 6. Two complete constraint automata (above), the product of the two automata before merging their bisimilar complement states (middle), and after merging their bisimilar complement states (bottom)

Definition 14 (Directional Data Constraints). *For a transition* $q \xrightarrow{N,g} p$ *, we define its directional data constraints* $g^\diamond$ *as the asymmetric relation built based on g and the type (mixed, source, sink) of nodes and state memory references that appear in g:*

$$g^\diamond = g^\Box \cup \{(x,?) \mid x \in N \cap \mathcal{N}^{mix} \wedge (x,y) \notin g^\Box\} \cup \{(?,x) \mid x \in N \cap \mathcal{N}^{mix} \wedge (y,x) \notin g^\Box\}$$

where the symbol "?" represents a function that generates fresh new unique special values that represent don't care *(i.e., no two don't care values produced by the "?" function are the same),*

$$g^\Box = \{(x,y) | (x,y) \in g^*, x \in \{\mathcal{N}^{src} \cup \mathcal{N}^{mix} \cup \mathcal{M}_g\} \wedge y \in \{\mathcal{N}^{snk} \cup \mathcal{N}^{mix} \cup \mathcal{M}'_g\}\}$$

and g^* *is the transitive closure of the related names* $g^\#$ *of the data constraints g.* □

The sets $\mathcal{M}_g$ and $\mathcal{M}'_g$ are defined in Section 3, following Definition 1. Observe that while g^* is symmetric, $g^\diamond$ is not. Moreover, $(x,y),(y,u) \in g^\diamond \implies y \in \mathcal{N}^{mix}$.

Definition 15 (Name set Subtraction Function). *The subtraction of a name set* $\mathcal{N}_1$ *from a complete constraint automaton* $\mathcal{A} = (Q, \mathcal{N}, \rightarrow, q_0, \mathcal{M})$*, denoted as* $\Lambda(\mathcal{A}, \mathcal{N}_1)$*, is a complete constraint automaton* $\mathcal{B} = (Q, \mathcal{N}', \rightarrow_r, q_0, \mathcal{M})$*, where* $\mathcal{N}' = \mathcal{N} \setminus \mathcal{N}^S$ *and* $\mathcal{N}^S = \{x | ((x,y) \in g^+ \vee (y,x) \in g^+) \implies x \in \mathcal{N}_1 \wedge y \in \mathcal{N}_1\}$*. The constraints set* g^+ *is constructed from the directional data constraints* $g^\diamond$ *of all transitions in* $\mathcal{A}$*:*

$$g^+ = \bigcup_{q \xrightarrow{N,g} p} g^\triangleright$$

where $g^\triangleright$ *is the transitive reduction of* $g^\diamond$*. The transition relation* $\rightarrow_r$ *is obtained from* $\rightarrow$ *by the following rule:*

$$\frac{q \xrightarrow{N,g} p}{q \xrightarrow{N \cap \mathcal{N}', g\restriction_{(\mathcal{N}' \cup \mathcal{M})}}_r p}$$

where $g \restriction_S$ *is obtained by substituting true for every atomic proposition in g that refers to a name* ***not*** *in* S *(causing the "removal" of those atomic propositions from g).* □

The transitive reduction of $g^\diamond$ yields the data-flows that occur from the firing source nodes and/or memory cells of the source state of a transition, through its firing mixed nodes, into its firing sink nodes and/or memory cells of its target state. The transitive reduction of a (finite) binary relation is not unique if it contains cycles. The algorithms for finding the transitive reduction of a relation allow removing any arbitrary tuple out of a relation to break the cycle and obtain a minimal relation [7].

Observe that cycles are possible in $g^\diamond$ only if they involve mixed nodes exclusively. Alternative $g^\triangleright$ reductions obtained by removing alternative tuples involved in such a cycle in a $g^\diamond$ yield data-flow paths that are equivalent under renaming of these (synchronously firing) mixed nodes. Thus, to obtain a $g^\triangleright$, we can remove tuples $(x,y) \in g^\diamond$ involved in a cycle arbitrarily if $x \notin \mathcal{N}_1$ or $y \notin \mathcal{N}_1$ (or both x and y are not in $\mathcal{N}_1$). If both x and y are in $\mathcal{N}_1$ then we remove either (x,y) or (y,x) arbitrarily, but not both.

Remark: removing a node name in a subtraction may change the classification of other remaining node names from mixed to sink or source. As it is a derivable detail, we skip this reclassification.

Definition 16 (Maximal Synchronized Names). *The function $\Phi : CCAS \times 2^{\mathcal{N}} \rightarrow 2^{2^{\mathcal{N}}}$ (where* CCAS *is the set of all complete constraint automata over the name set $\mathcal{N}$) designates the sets of* maximal synchronized names *of a complete constraint automaton excluding the synchronization between names of a given set $\mathcal{N}_1 \subseteq \mathcal{N}$. For a complete constraint automaton $\mathcal{A} = (Q, \mathcal{N}, \rightarrow, q_0, \mathcal{M})$, every $\mathcal{S} \in \Phi(\mathcal{A}, \mathcal{N}_1)$ is a maximal subset of $\mathcal{N}$ such that for all transitions $q \xrightarrow{N,g} p$ of $\mathcal{A}$ and every pair of (not necessarily distinct) X and Y in $\mathcal{S}$ where $X \notin \mathcal{N}_1 \vee Y \notin \mathcal{N}_1$, $X \in N \implies Y \in N$.* □

Intuitively, the node names in a $\mathcal{S} \in \Phi(\mathcal{A}, \mathcal{N}_1)$ have the property that either all or none of them appear in the name set of a transition in $\mathcal{A}$.

By subtraction, some of the synchronization and data-flow relations are removed and this may produce automata that can be partitioned into other automata with disjoint sets of node names and memories.

Definition 17 (Automata Partitioning Function). *The partitioning function $\Gamma : CCAS \times 2^{2^{\mathcal{N}}} \rightarrow 2^{CCAS}$ (where* CCAS *is the set of all complete constraint automata over the name set $\mathcal{N}$) partitions a complete constraint automaton into a finite set of complete constraint automata, preserving a given maximal synchronized names set. Specifically, for a complete constraint automaton $\mathcal{A} = (Q, \mathcal{N}, \rightarrow, q_0, \mathcal{M})$ and a maximal synchronized names set $\mathcal{Z}$, $\Gamma(\mathcal{A}, \mathcal{Z})$ partitions $\mathcal{A}$ into a finite number of complete constraint automata $\mathcal{A}_i = (Q, \mathcal{N}_i, \rightarrow_i, q_0, \mathcal{M}_i)$, for $0 < i <= n$. The disjoint non-empty sets $\mathcal{N}_1, ...\mathcal{N}_n$ and $\mathcal{M}_1, ...\mathcal{M}_n$ are obtained as the maximal number of partitions of the sets $\mathcal{N}$ and $\mathcal{M}$, respectively, such that:*

1. *$\mathcal{N}_i$ contains every $x \in \mathcal{N}$ that is related to a $y \in \mathcal{N}_i$ in every $G \in \mathcal{G}$,*
2. *for all $\mathcal{S} \in \mathcal{Z}, \mathcal{S} \subseteq \mathcal{N}_i$ for $0 < i <= n$, and*
3. *$\mathcal{M}_i$ contains every $x \in \mathcal{M}$ that is related to a $m \in \mathcal{M}_i$ or a $y \in \mathcal{N}_i$ in every $G \in \mathcal{G}$,*

where

$$\mathcal{G} = \bigcup_{q \xrightarrow{N,g} p} \{\{r\} \mid r \in (g^{\#})^*\}$$

and $(g^{\#})^*$ is the transitive closure of the related names $g^{\#}$ of the data constraints g of each transition in $\rightarrow$.

The transition relations $\rightarrow_i$ are derived from $\rightarrow$ by the following rule:

$$\frac{q \xrightarrow{N,g} p}{q \xrightarrow{N \cap \mathcal{N}_i, g \restriction_{\mathcal{N}_i \cup \mathcal{M}_i}}_i p}$$

*where $g \restriction_S$ is obtained by substituting true for every atomic proposition in g that refers to a name **not** in S (causing the "removal" of those atomic propositions from g).* □

Theorem 1. *If* $\mathcal{A}_1 \bigodot \mathcal{A}_2 = \Upsilon(\xi_{\mathcal{N}_r})$ *and* $\mathcal{A}_1 \bigodot \overline{\mathcal{A}_2} = \mathcal{A}_1$ *where* $\mathcal{A}_1 \not\simeq \Upsilon(\xi_{\mathcal{N}_{\mathcal{A}_1}}) \wedge \mathcal{A}_2 \not\simeq \Upsilon(\xi_{\mathcal{N}_{\mathcal{A}_2}})$ *then* A_1 *and* A_2 *share a "sub-automaton" and its inverse, i.e., the* CCA B, E *and* D *exist such that* $\mathcal{A}_1 = B \bigodot E$ *and* $\mathcal{A}_2 = D \bigodot \overline{E}$ *and* $\mathcal{N}_E = (\mathcal{N}_{\mathcal{A}_1} \cup \mathcal{N}_{\mathcal{A}_2}) \setminus \mathcal{N}_r$ *and* $(\mathcal{N}_r = \mathcal{N}_B \cup \mathcal{N}_D)$. □

Based on Theorem 1, we define the product-automaton for complete constraint automata as follow:

Definition 18 (Product-automaton for *CCA*)
The product-automaton of the two complete constraint automata $\mathcal{A}_1$ *and* $\mathcal{A}_2$ *is* $\mathcal{A}_1 \bowtie \mathcal{A}_2$*:*

$$\mathcal{A}_1 \bowtie \mathcal{A}_2 = (\bigodot_{i=1}^{n} L_{1_i}) \bigodot (\bigodot_{i=1}^{m} L_{2_i})$$

where

$$(L_{1_1},\ L_{1_2},\ ...,\ L_{1_n}) = \Gamma(\Lambda(\mathcal{A}_1, s), \Phi(\mathcal{A}_1, s))$$

$$(L_{2_1},\ L_{2_2},\ ...,\ L_{2_m}) = \Gamma(\Lambda(\mathcal{A}_2, s), \Phi(\mathcal{A}_2, s))$$

The symbol s *refers to the name set as below:*

$$s = \begin{cases} (\mathcal{N}_{\mathcal{A}_1} \cup \mathcal{N}_{\mathcal{A}_2}) \setminus \mathcal{N}_r & \textit{if}\ \mathcal{A}_1 \bigodot \mathcal{A}_2 = \Upsilon(\xi_{\mathcal{N}_r}) \ \wedge\ \mathcal{A}_1 \bigodot \overline{\mathcal{A}_2} = \mathcal{A}_1 \\ \emptyset & \textit{otherwise} \end{cases}$$ □

Note 1: $\mathcal{A}_1 \bowtie \mathcal{A}_2 = \mathcal{A}_1 \bigodot \mathcal{A}_2$ if $\mathcal{A}_1 \bigodot \mathcal{A}_2 \neq \Upsilon(\xi_{\mathcal{N}_r})$ or $s = \emptyset$.
Note 2: $\mathcal{A}_1 \bigodot \mathcal{A}_2 = \Upsilon(\xi_{\mathcal{N}_r}) \wedge \mathcal{A}_1 \bigodot \overline{\mathcal{A}_2} = \mathcal{A}_1 \Rightarrow \overline{\mathcal{A}_1} \bigodot \mathcal{A}_2 = \mathcal{A}_2$ where $\mathcal{A}_1 \not\simeq \Upsilon(\xi_{\mathcal{N}_{\mathcal{A}_1}}) \wedge \mathcal{A}_2 \not\simeq \Upsilon(\xi_{\mathcal{N}_{\mathcal{A}_2}})$.

Due to space limitation, in the sequel, we omit the details and the proofs.

Lemma 4. ***(Name set Un-hiding)*** *For the complete constraint automaton* $\mathcal{A}_1$*,* $\mathcal{A}_2$*, and* $\mathcal{B}$*, where* $\mathcal{B} = \mathcal{A}_1 \bowtie \mathcal{A}_2$*:*

$$\mathcal{B} \bowtie \mathcal{A}_1 \simeq (\mathcal{A}_1 \bowtie \mathcal{A}_2) \bowtie \mathcal{A}_1 \simeq (\mathcal{A}_1 \bowtie \mathcal{A}_1) \bowtie \mathcal{A}_2 \simeq (\mathcal{A}_1 \bowtie \mathcal{A}_2) \simeq \mathcal{B}$$

□

Lemma 5. ***(Self Product)*** *For the complete constraint automaton* $\mathcal{A}$*,* $\mathcal{A} = \mathcal{A} \bowtie \mathcal{A}$. □

5 Decomposition Example

To decompose a constraint automaton $\mathcal{A}$ based on a given constraint automaton $\mathcal{B}$, we only compute the product of the completion of $\Upsilon(\mathcal{A})$ with the inverse of the completion of $\Upsilon(\mathcal{B})$: The complete constraint automaton $\mathcal{C} = \Upsilon(\mathcal{A}) \bowtie \overline{\Upsilon(\mathcal{B})}$ is the result of extracting $\mathcal{B}$ from $\mathcal{A}$. Based on Theorem 1, if $\mathcal{B}$ is a part of the constraint automaton $\mathcal{A}$, the result of $\Upsilon(\mathcal{A}) \bigodot \overline{\Upsilon(\mathcal{B})}$ must be $\Upsilon(\xi_{\mathcal{N}_\mathcal{A} \setminus \mathcal{N}_\mathcal{B}})$, and $\Upsilon(\mathcal{A}) \bigodot \Upsilon(\mathcal{B})$ must be $\Upsilon(\mathcal{A})$.

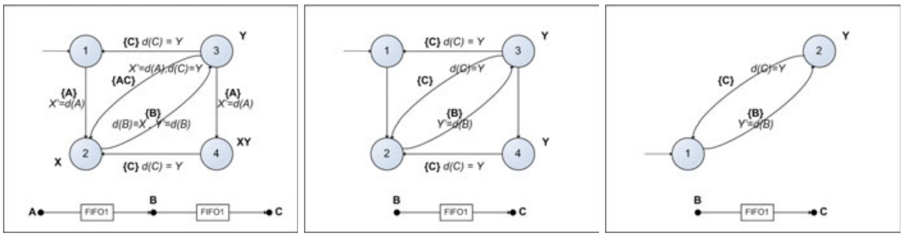


Fig. 7. The left figure shows constraint automaton $\mathcal{A}$ for two *FIFO1* channels composed in a simple Reo circuit. The middle figure shows the constraint automaton $\mathcal{A}$ after name subtraction. The right figure shows the constraint automaton $\mathcal{A}$ after merging the states with the same state descriptor. For simplicity these figures show only the real states of the automata.

Figure 7 shows a simple Reo circuit with two *FIFO1* channels and its constraint automaton with state memory $\mathcal{A}$. We decompose $\mathcal{A}$ by extracting a *FIFO1* channel from it.

$$\mathcal{A} = (\{1,2,3,4\}, \mathcal{N}_{\mathcal{A}} = \{A,B,C\}, \rightarrow_{\mathcal{A}}, \mathcal{M}_{\mathcal{A}} = \{X,Y\})$$

$$\mathcal{B} = (\{1,2\}, \mathcal{N}_{\mathcal{B}} = \{A,B\}, \rightarrow_{\mathcal{B}}, \mathcal{M}_{\mathcal{B}} = \{X\})$$

$$\mathcal{C} = (\{1,2\}, \mathcal{N}_{\mathcal{C}} = \{B,C\}, \rightarrow_{\mathcal{C}}, \mathcal{M}_{\mathcal{C}} = \{Y\})$$

If we use the simple product-automaton we obtain the following result:

$$\varUpsilon(\mathcal{A}) \bigodot \overline{\varUpsilon(\mathcal{B})} = \varUpsilon(\xi_{\mathcal{N}_{\mathcal{A}} \backslash \mathcal{N}_{\mathcal{B}}})$$

The result is the null automaton and we obtain the $\mathcal{A}$ as a result of the following computation:

$$\varUpsilon(\mathcal{A}) \bigodot \varUpsilon(\mathcal{B}) = \varUpsilon(\mathcal{A})$$

According to the above results, we must apply subtraction and partitioning. We remove $\mathcal{N}_{\mathcal{B}}$ from $\mathcal{B}$ using the subtract function $\Lambda(\mathcal{B}, \mathcal{N}_{\mathcal{B}})$, obtaining the null automaton which cannot be partitioned.

We then remove $\mathcal{N}_B$ from $\mathcal{A}$ using the subtract function $\Lambda(\mathcal{A}, \mathcal{N}_B)$ and with $\Phi(\mathcal{A}, \mathcal{N}_B) = \{\{A\},\{B\}\}$, we obtain the automaton shown in Figure 7, which cannot be partitioned.

The final result of the product is the second *FIFO1* channel in Figure 7 (after merging the states with the same state descriptor).

As a larger case study, we worked on the Reo circuit and constraint automata for orchestrating three components in a service oriented application. We had the constraint automata for the three components in Figure 8, and the required behaviour of the system in Figure 9. We extracted the automata for the three components from the automata of their required behaviour and obtained the automaton for their orchestration and then construct the Reo circuit using the same approach.

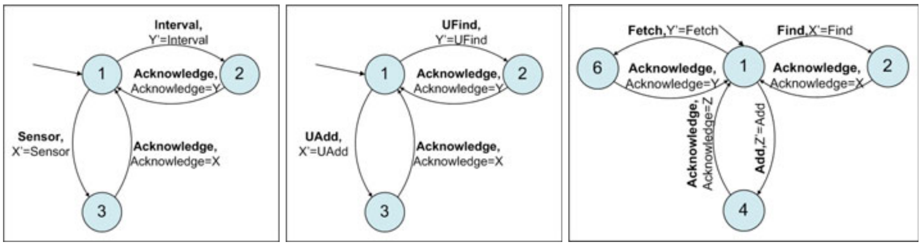


Fig. 8. Figures from left to right show an automaton for the service dynamic manager component, the UDDI Discovery component, and the user dynamic manager component

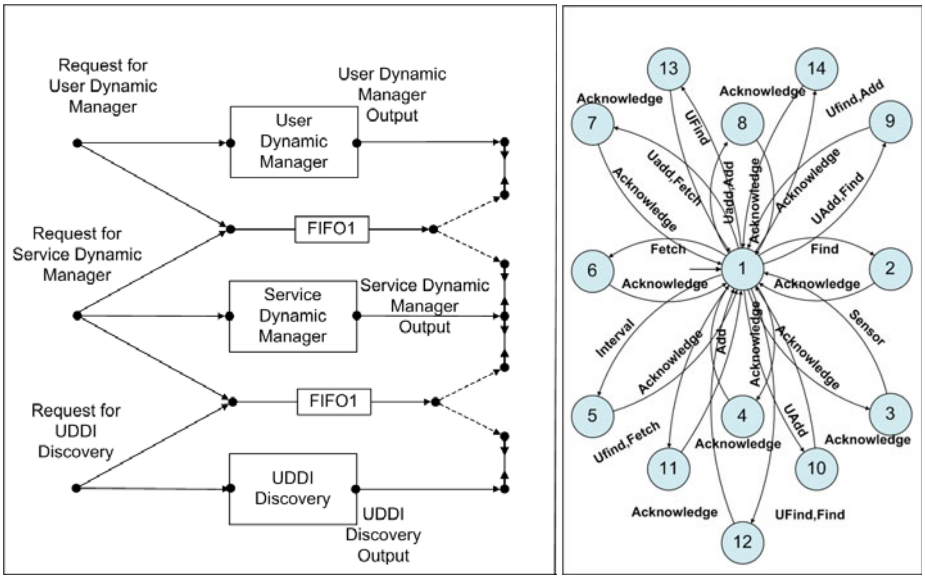


Fig. 9. Architecture based on the Reo circuit and required composed system and its automaton

6 Related Work

The problem of decomposing complex systems into simpler components is one of the fundamental problems in both science and engineering. The particular case of interacting finite state automata is considered in many disciplines in computer science, such as distributed computing, hardware realizations, distributed AI, and behavior-based robotics - to mention a few. To accomplish this, automata decomposition was first introduced by Krohn and Rhodes in [8], and Eilenberg introduced Holonomy Decomposition in [9]. The holonomy method appears to be relatively efficient and has been implemented by Egri-Nagy in [10]. A specific automata set with a binary product operation form a semigroup in [8] and [9]. Each of these decomposition disciplines works on a specific language and takes advantage of how the combination of some primitive automata yields that language.

Schützenberger was the first to establish the semigroup as the fundamental mathematical structure of ordinary finite automata [11]. Actually, the proper algebraic structure for finite automata and regular languages is the Kleene algebra, a semi-ring with an extra unary operation [12].

The automata we consider in this paper are different from the classical finite automata as they have constraints and memory cells. The richness of this language allows us to move from semigroups to Abelian groups, thus introducing an inverse automaton for each complete constraint automaton.

Synthesis of Reo circuits was first introduced in [13] where the authors showed how to construct a Reo circuit from schedule-data stream. Another synthesis method was introduced by Koehler and Clarke in [14] on port automata. Basically, a port automaton is a constraint automaton without any data constraint. They showed that port automata can be synthesized using only the merger primitive. This work is in the same line as Vuillemin and Gama in [15] where they showed that there is a normal form for regular languages which can be generated by using only the primitive *XOR*.

For decomposition or synthesis, existing similar works generally expand the automata, which typically transforms a simple automaton to a large automaton. They decompose each automaton to a specific set of predefined primitive automata. Moreover, they use the inverse of the language generated by the automata and not the automata themselves. These works abstract additional constraints or conditions on the automata.

We show that a set of *CCA* on a specific name set with a binary product operator are invertible and we present a formal definition to obtain the inverse of complete constraint automata. We can then use this inverse to decompose any complex automata into an arbitrary set of given components.

7 Conclusion and Future Work

In this work we defined the inverse of a constraint automaton which will be used to build an algebraic system. To do this, we have extended constraint automata to complete constraint automata.

We can show that a set of complete constraint automata (*CCAS*) over a given name set, together with the product operation form an Abelian Group in an algebraic system.

We introduced rules and lemmas that are used in decomposition of constraint automata. These rules can help us to perform model-driven development, reverse engineering, and automatic programming. Starting from a given model for the system, and a set of reusable off-the-shelf components, all specified in the form of constraint automata, we can iteratively extract the *CA* of components from the *CA* model of a desired system and obtain the *CA* of the necessary missing building blocks, as the "remainder" in this operation. Currently, we are working on building a set of rules to show the criteria for selecting the suitable components in each step. These criteria can be customized based on the availability of components, their cost, speed and other relevant parameters.

Moreover, we will investigate the interesting features of Abelian groups that can be used for different goals. Lemma 1 of the paper shows that the completion function is an isomorphism. This implies that it should be possible to apply a variant of the inverse function proposed in this paper directly on constraint automata. Hence, this work serves as an exercise to build the rationale towards simpler techniques which can be the basis for developing more efficient decomposition and synthesis tools.

References

1. Arbab, F.: Reo: a channel-based coordination model for component composition. Mathematical Structures in Computer Science 14, 329–366 (2004)
2. Arbab, F., Baier, C., Rutten, J.J.M.M., Sirjani, M.: Modeling component connectors in Reo by constraint automata (extended abstract). Electr. Notes Theor. Comput. Sci. 97, 25–46 (2004)
3. Mehta, N.R., Medvidovic, N., Sirjani, M., Arbab, F.: Modeling behavior in compositions of software architectural primitives. In: ASE, pp. 371–374. IEEE Computer Society, Los Alamitos (2004)
4. Razavi, N., Sirjani, M.: Using Reo for formal specification and verification of system designs. In: MEMOCODE, pp. 113–122. IEEE, Los Alamitos (2006)
5. Meng, S., Arbab, F.: Web services choreography and orchestration in Reo and constraint automata. In: Cho, Y., Wainwright, R.L., Haddad, H., Shin, S.Y., Koo, Y.W. (eds.) SAC, pp. 346–353. ACM, New York (2007)
6. Arbab, F., Rutten, J.J.M.M.: A coinductive calculus of component connectors. In: Wirsing, M., Pattinson, D., Hennicker, R. (eds.) WADT 2003. LNCS, vol. 2755, pp. 34–55. Springer, Heidelberg (2003)
7. Aho, A.V., Garey, M.R., Ullman, J.D.: The transitive reduction of a directed graph. SIAM J. Comput. 1, 131–137 (1972)
8. Krohn, K., Rhodes, J.: Algebraic theory of machines. I. prime decomposition theorem for finite semigroups and machines. Transactions of the American Mathematical Society, vol. 116, pp. 450–464. ACM, New York (1965)
9. Eilenberg, S.: Automata, languages and machines (1976)
10. Egri-Nagy, A., Nehaniv, C.L.: Algebraic hierarchical decomposition of finite state automata: Comparison of implementations for Krohn-Rhodes theory. In: Domaratzki, M., Okhotin, A., Salomaa, K., Yu, S. (eds.) CIAA 2004. LNCS, vol. 3317, pp. 315–316. Springer, Heidelberg (2005)
11. Schützenberger, M.: On the definition of a family of automata. Information and Control, 245–270 (1961)
12. Kozen, D.: On Kleene algebras and closed semirings. In: Proc. Mathematical Foundations in Computer Science, vol. 452, pp. 26–47 (1990)
13. Arbab, F., Baier, C., de Boer, F.S., Rutten, J.J.M.M., 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)
14. Koehler, C., Clarke, D.: Decomposing port automata. In: Shin, S.Y., Ossowski, S. (eds.) SAC, pp. 1369–1373. ACM, New York (2009)
15. Vuillemin, J., Gama, N.: Compact normal form for regular languages as xor automata. In: Maneth, S. (ed.) CIAA 2009. LNCS, vol. 5642, pp. 24–33. Springer, Heidelberg (2009)

Graph Representation of Sessions and Pipelines for Structured Service Programming*

Roberto Bruni[1], Zhiming Liu[2], and Liang Zhao[1,2,**]

[1] Department of Computer Science, University of Pisa, Italy
[2] United Nations University - International Institute for Software Technology, P.O. Box 3058, Macao
liang@iist.unu.edu

Abstract. Graph transformation techniques, and the Double-Pushout approach in particular, have been successfully applied in the modeling of concurrent systems. In this area, a research thread has addressed the definition of concurrent semantics for process calculi. In this paper, we show how graph transformation can cope with advanced features of service-oriented process calculi, such as several logical notions of scoping (like sessions and pipelines) together with the interplay between linking and containment. This is illustrated by encoding CaSPiS, a recently proposed process calculus with such sophisticated features. We show how to represent the congruence and reduction relations between CaSPiS processes by exploiting concurrent graph transformations over hierarchical graphs.

Keywords: process calculus, hierarchical graph, graph transformation.

1 Introduction

Process calculi are a flexible mathematical formalism that provides a convenient abstraction for the study of concurrent systems, in the same way as λ-calculus lays the foundation of sequential computation. The main ingredients of process calculi are: an algebra (i.e. a signature and a set of structural congruence axioms) of computational entities, called *processes* with primitives for communication, parallel composition, etc., and an operational semantics modelling the evolution of processes either in terms of a labelled transition system or as a reduction system, that poses the basis for studying several notions of behavioural equivalence over processes.

Process calculi have become quite mature in studying traditional concurrent and communicating systems [11,14], and even advanced to specification and verification of mobile systems [15]. However, these traditional process calculi does not match certain advanced features of service-oriented computing like the nested scoping of sessions or pipelining workflows, or the interplay between linking and containment. Though there exist attempts of using π-calculus [15] as a model of service systems [13,8], the encoding of channels, sessions and pipelines are quite low level and different first-class

* The work is supported by the project GAVES funded by Macau Science and Technology Development Fund, NSFC 60970031, 973 program 2009CB320702, and STCSM 08510700300 and by the Italian MIUR project IPODS (PRIN 2008).

** Correspondence author.

aspects in SOC are mixed up and obfuscated. The low level communication primitives of π-calculus make the analysis, e.g. about safe interactions, too complex and even infeasible.

The Service Centered Calculus (SCC) [1] introduces *service definition*, *service invocation* and *session handling* as first class modeling elements, so as to model service systems at a better level of abstraction. However, SCC has a rudimentary mechanism for handling session closure, and it has no mechanism for orchestrating values arising from different activities. These aspects have been improved in the *Calculus of Session and Pipelines* (CaSPiS) [2]. CaSPiS still supports the important features of SCC with respect to service autonomy, client-service interaction and orchestration. However, the notions of session and pipelining play a more central role. In CaSPiS, a session has two sides (or participating processes) and it is equipped with protocols followed by each side during an interaction between the two sides. A pipeline in CaSPiS permits orchestrating the flow of data produced by different sessions. The concept of pipeline is inspired by Orc [16], a basic and elegant programming model for structured orchestration of services. A structured operational semantics of CaSPiS is given in [2] based on labeled transitions. It does yet have a simpler and compact reduction semantics [3], on which we focus, that handles silent actions of processes in the labeled transition system.

As illustrated by a large body of literature, graphs and graph transformations provide useful insights into distributed, concurrent and mobile systems [10,12,9]. Following this direction, we are going to define a graph-based concurrent semantics for CaSPiS. This can help, for example, to record causal dependencies between interactions and exploit such information for detecting the possible source of faults and misbehaviors. In order to succeed we need to address two issues. The first is that sessions and pipelines introduce a strong hierarchical nature to a service oriented system in both of its static structure and dynamic behavior. The hierarchical structure also changes during the evolution of the system, due to dynamic creation of sessions by invocations of services, and dynamic creation of processes in pipelines. Therefore we must deal with hierarchical graphs. The second is that the graph transformation semantics must be "compatible" with the existing interleaving one.

In this paper, we propose a hierarchical graph representation of service oriented systems and show how to use graph transformation rules to study their behaviors. More precisely, our first contribution is to set up a model of hierarchical graphs by exploiting a suitable graph algebra. In this model, graph transformations are studied following the well-known Double-Pushout (DPO) approach [7]. Then, we map CaSPiS processes to hierarchical graphs in the graph algebra and define a graph transformation system with two sets of graph transformation rules to represent the congruence relation and reduction semantics of CaSPiS, respectively. Finally, we provide the soundness and completeness results of such representation with respect to the congruence relation as a main result and conjecture a similar result to hold for the reduction semantics. Our framework can be extended to deal with persistent services, i.e. services always available for invocations, without compromising the soundness and completeness results.

There is some work that also aims at providing a graph model for SOC. In [5], states of service systems are interpreted as terms of a graph algebra that supports names, name restrictions and design hierarchy. We adopt the grammar of the algebra, but provide a different semantic model where hierarchy is realized by a special kind of edges, called *abstract edges*. With such a model, we are able to define graph transformation rules in the DPO form that change the hierarchical structure of a graph, i.e. through adding or removing the corresponding abstract edges.

An extension of the work [5] is presented in [4] where the behaviors of processes are also studied. The authors defined standard forms of graph transformation rules for reductions of processes, while each rule only deals with the case that the reduction occurs in a specific context. To handle reductions in all possible contexts, an infinite number of rules would be needed. Compared with this work, our graph model is based on the DPO approach which enables us to define the behaviors of processes through a finite number of rules.

We introduce the calculus CaSPiS in the next section, and our graph model in Section 3. In Section 4, we give the graph representation of CaSPiS processes and define the graph transformation system. Section 5 is a small example to illustrate the application of the graph transformation rules, and Section 6 draws some conclusions.

2 The Calculus CaSPiS

This section introduces the key notions of the service-oriented calculus CaSPiS [2]. Let $\mathcal{S}$, $\mathcal{R}$, $\mathcal{V}$ and $\mathcal{C}$ be four disjoint infinite sets, respectively of service names, session names, variables and constants. Variables and constants are called *values*.

The simplest process is the inactive process $\mathbf{0}$ (nil). A process P can be prefixed by an *abstraction* $(?x)$ that is ready to receive a value and assign it to each occurrence of the variable x occurring in P; a *concretion* $\langle V\rangle$ that generates an output value V; or a *return* $\langle V\rangle^{\uparrow}$ that returns a value V to the environment outside the current session.

The standard parallel composition $P|Q$ is allowed. The choice operator "$+$" is limited to the nil process, as well as those processes prefixed with abstractions, concretions and returns, called the *prefixed processes*, i.e. only guarded sums are allowed.

A service is declared by a service definition $s.P$ and used by the environment through a service invocation $\overline{s}.Q$. A participant process of a session is represented by $r \rhd P$, where r is a session name, and P is the *protocol* process this participant follows. In CaSPiS, a session r can have only two participants that interact with each other. They are also called the *two sides* of the session.

A process P can be pipelined with another process Q, denoted by $P > Q$, so that P can keep producing values for Q to consume. Service names, session names and variables can be restricted, in a way like the π-calculus [15] by $(\nu n)P$. This restricts all the occurrences of the name n within P, and P is called the scope of the restriction.

Definition 1 (CaSPiS process). *A process is a term generated by the syntax:*

$$\begin{array}{lrl} \textit{Process} & P,Q ::= & M \mid P|Q \mid s.P \mid \overline{s}.P \mid r\rhd P \mid P>Q \mid (\nu n)P \\ \textit{Sum} & M ::= & \mathbf{0} \mid \pi P \mid M+M \\ \textit{Prefix} & \pi ::= & (?x) \mid \langle V\rangle \mid \langle V\rangle^{\uparrow} \end{array}$$

where $s \in \mathcal{S}$, $r \in \mathcal{R}$, $x \in \mathcal{V}$, $V \in \mathcal{V} \cup \mathcal{C}$ *and* $n \in \mathcal{S} \cup \mathcal{R} \cup \mathcal{V}$.

We remark that the session construct $r \rhd P$ is a runtime syntax: it should not be used to model the initial state of a system, but can be dynamically generated upon service

$$\begin{array}{rclrcll}
(P|P')|P'' & \equiv_c & P|(P'|P'') & (\nu n)\mathbf{0} & \equiv_c & \mathbf{0} & \\
P|P' & \equiv_c & P'|P & (\nu n)(\nu n')P & \equiv_c & (\nu n')(\nu n)P & \\
(M+M')+M'' & \equiv_c & M+(M'+M'') & P|(\nu n)Q & \equiv_c & (\nu n)(P|Q) & \text{if } n \notin \text{fn}(P) \\
M+M' & \equiv_c & M'+M & (\nu n)Q > P & \equiv_c & (\nu n)(Q > P) & \text{if } n \notin \text{fn}(P) \\
M+\mathbf{0} & \equiv_c & M & r \rhd (\nu n)P & \equiv_c & (\nu n)(r \rhd P) & \text{if } n \neq r
\end{array}$$

Fig. 1. Cases of congruence

invocations. We omit $\mathbf{0}$ in a prefixed term and write, for example, $(?x)\langle x\rangle$ for $(?x)\langle x\rangle\mathbf{0}$. A name n occurring in a process is *free* if it is not bound by either an abstraction $(?n)$ or a restriction (νn), and $\text{fn}(P)$ denotes the set of free names of P. We do not distinguish between processes that are alpha-convertible, such as $(?x)\langle x\rangle\langle z\rangle$ and $(?y)\langle y\rangle\langle z\rangle$. In addition, we have a set of *structural congruence rules* among processes, given in Fig. 1. They are standard monoidal laws and rules for moving restrictions forward.

2.1 Operational Semantics in Terms of Reduction

The basic behavior of a process P is the communication and synchronization (called interactions) between its sub-processes. After an interaction, P evolves to another process Q. A step of such a change is called a *reduction*, denoted as $P \rightarrow Q$.

The behaviors of the prefixed processes, the sum "+", parallel composition, and restriction are similar to those in a traditional process calculus, such as π-calculus [15].

A service definition process $s.P$ and service invocation process $\overline{s}.Q$ synchronize on the service s and its corresponding invocation $\overline{s}$. After offering the service s, $s.P$ evolves to a session process $r \rhd P$ with a fresh session name r. Symmetrically, after the service invocation $\overline{s}$, $\overline{s}.Q$ becomes a session process $r \rhd Q$ of the same session name r. For example, $s.P|\overline{s}.Q \rightarrow (\nu r)(r \rhd P|r \rhd Q)$.

The session name r will bound the communication scope of P and Q. The freshness of r guarantees that other processes cannot interfere with the interaction between P and Q. When a session r starts, the protocols P and Q of the session sides $r \rhd P$ and $r \rhd Q$ become active, i.e. they are ready to produce and receive values from each other. This is shown by the example $r \rhd (?x)P|r \rhd \langle V\rangle Q \rightarrow r \rhd P[V/x]|r \rhd Q$.

A pipelined process $P > Q$ behaves as P but keeps the new state of P pipelined with Q, until P produces a value. When P produces a value, a new instance of Q is created, that consumes the value produced by P and then runs in parallel with the pipeline. For example: $\langle V\rangle P > (?x)Q \rightarrow (P > (?x)Q)|Q[V/x]$.

Context. The formal definition of a reduction needs the notion of *context*, which is a process expression with some "holes". Specifically, a context with k holes is a process term $C(X_1,\ldots,X_k)$ defined in Definition 1, but with *processes variables* $X_1,\ldots,X_k$ in it. In this paper, we only need contexts with one or two holes, and we omit the process variables and denote a context as $C[\cdot]$ or $C[\cdot,\cdot]$. A context is called *static* if none of its holes occurs in the scope of a *dynamic operator*, which is either a service definition $s.[\cdot]$, a service invocation $\overline{s}.[\cdot]$, a sum $[\cdot]+M$ or $M+[\cdot]$, a prefix $\pi[\cdot]$, or the right-hand side of a pipeline $P > [\cdot]$. A context is *session-immune* if its hole(s) does not occur in

(Sync)	$P \equiv_c C[s.P_1,\ \bar{s}.P_2]$ $Q \equiv_c (\nu r)C[r \rhd P_1,\ r \rhd P_2]$ r fresh for $C[\cdot,\cdot], P_1, P_2$
(S-Sync)	$P \equiv_c C[r \rhd (P' \mid (\langle V\rangle P_1 + M_1)),\ r \rhd C_1[(?x)P_2 + M_2]]$ $Q \equiv_c C[r \rhd (P' \mid P_1),\ r \rhd C_1[P_2\sigma]]$
(S-Sync-Ret)	$P \equiv_c C[r \rhd (P' \mid r' \rhd C_2[\langle V\rangle^{\uparrow} P_1 + M_1]),\ r \rhd C_1[(?x)P_2 + M_2]]$ $Q \equiv_c C[r \rhd (P' \mid r' \rhd C_2[P_1]),\ r \rhd C_1[P_2\sigma]]$
(P-Sync)	$P \equiv_c C_0[(P' \mid (\langle V\rangle P_1 + M_1)) > ((?x)P_2 + M_2)]$ $Q \equiv_c C_0[P_2\sigma \mid ((P' \mid P_1) > ((F)P_2 + M_2))]$
(P-Sync-Ret)	$P \equiv_c C_0[(P' \mid r \rhd C_2[\langle V\rangle^{\uparrow} P_1 + M_1]) > ((?x)P_2 + M_2)]$ $Q \equiv_c C_0[P_2\sigma \mid ((P' \mid r \rhd C_2[P_1]) > ((?x)P_2 + M_2))]$

Fig. 2. Cases of reduction

the scope of a session, and *restriction-immune* if its hole(s) does not occur in the scope of a restriction. A 2-hole context is called *restriction-balanced* if the holes occur in the same restriction environment. For example, $(\nu n)[\cdot] \mid r \rhd [\cdot]$ is a static context that is not restriction-balanced, since the restriction (νn) bounds the first hole but not the second.

Reduction Rules. Following the discussion about the informal behavior of processes, we summarize the reduction rules in Fig. 2 for service definition, service invocation, session and pipelined processes, where each rule shows a pair of processes P and Q such that $P \rightarrow Q$. In Fig. 2, σ denotes the substitution $[V/x]$; $C_0[\cdot]$ is static; $C[\cdot,\cdot]$ is static and restriction-balanced; $C_1[\cdot]$ and $C_2[\cdot]$ are static, session-immune and restriction-immune. There is no rule that allows a reduction to take place in a non-static context.

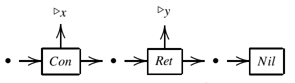
Let us consider the process $Ser \mid (Cl > (?y)P)$, where $Ser = req.(\nu\ell)(\langle \ell\rangle + \langle \mathsf{null}\rangle)$ is a service to allocate new resources (if available), $Cl = \overline{req}.(?x)\langle x\rangle^{\uparrow}$ is a client of Ser and P is a generic process. Then the above process can evolve as illustrated below:

$$\begin{aligned}
Ser \mid (Cl > (?y)P) &\rightarrow (\nu r)(\, r \rhd (\nu\ell)(\langle\ell\rangle + \langle\mathsf{null}\rangle) \mid (r \rhd (?x)\langle x\rangle^{\uparrow} > (?y)P)) && \text{by (Sync)}\\
&\equiv_c (\nu r)(\nu\ell)(\, r \rhd \langle\ell\rangle + \langle\mathsf{null}\rangle \mid (r \rhd (?x)\langle x\rangle^{\uparrow} > (?y)P))\\
&\rightarrow (\nu r)(\nu\ell)(\, r \rhd \mathbf{0} \mid (r \rhd \langle\mathsf{null}\rangle^{\uparrow} > (?y)P)) && \text{by (S-Sync)}\\
&\rightarrow (\nu r)(\nu\ell)(\, r \rhd \mathbf{0} \mid (r \rhd \mathbf{0} > (?y)P) \mid P[\mathsf{null}/y]) && \text{by (P-Sync-Ret)}
\end{aligned}$$

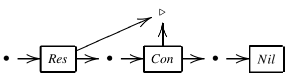
Note that, as $r \rhd \mathbf{0}$ is clearly inert and therefore also $r \rhd \mathbf{0} > (?y)P$ is inert, then the reached process amounts essentially to $P[\mathsf{null}/y]$. An analogous computation could have led (up to the presence of inert processes) to the process $(\nu\ell)P[\ell/y]$.

3 Algebra of Hierarchical Graphs

A CaSPiS process can be represented as a (hyper)graph. For example in Fig. 3(a), the graph of $P = \langle x\rangle\langle y\rangle^{\uparrow}$ shows that P generates a value x, returns a value y and then becomes the nil process. The unnamed • nodes represent the states of the control flow, and the nodes $\rhd x$ and $\rhd y$ show that x and y are values generated and returned by the *concretion* and *return* edges, respectively. The graph in Fig. 3(b) shows that process Q generates a value z but this value is restricted, thus invisible from outside. A restricted



(a) $P = \langle x \rangle \langle y \rangle^{\uparrow}$



(b) $Q = (\nu z)\langle z \rangle$

Fig. 3. Graph representations of processes

value node is therefore not named in the graph. These graphs are called *hypergraphs*, since an edge can be associated with one or more nodes. A hypergraph shows the control flow and data flow, as well as the structure of a process, through different types of nodes, such as $\triangleright$ and $\bullet$, and different types of edges, such as *Ret*, *Res* and *Con*.

3.1 Graph Grammar

We use a graph algebra [5] to specify and study the algebraic properties of hypergraphs, in order to study CaSPiS processes. Our vocabulary includes a set of node names $\mathcal{N}$ and set of edge labels $\mathcal{L}$. We also use $\overline{x}$ to denote a sequence, $\overline{x}[i]$, $\{\overline{x}\}$ and $|\overline{x}|$ to denote the i-th member, the set of members and the length of the sequence, respectively.

Definition 2 (Graph term). *A graph term is generated by the grammar*

$$G ::= \mathbf{0} \mid x \mid l(\overline{x}) \mid G|G \mid (\nu x)G$$

where $x \in \mathcal{N}$ and $l \in \mathcal{L}$.

Term $\mathbf{0}$ represents the empty graph, x is the graph of only one node whose name is x, $l(\overline{x})$ is the graph of an l-labeled edge attached to nodes $\overline{x}$ through its *tentacles*, $G_1|G_2$ roughly represents the union[1] of G_1 and G_2 and $(\nu x)G$ is a *restriction* that binds the name x in G so that it is invisible outside G.

A process P of CaSPiS can be represented by a graph term, denoted by $[\![P]\!]$. The graph terms of processes P and Q in Fig. 3 are respectively written as

$$\begin{aligned}[\![P]\!] &= (\nu p)(\nu p_1)(\nu p_2)(Con(p,x,p_1)|Ret(p_1,y,p_2)|Nil(p_2))\\ [\![Q]\!] &= (\nu p)(\nu p_1)(\nu p_2)(\nu z)(Res(p,z,p_1)|Con(p_1,z,p_2)|Nil(p_2)).\end{aligned}$$

For a graph term G, we denote its hypergraph as $\mathcal{H}(G)$ and called it the *hypergraph of* G, and $\mathcal{H}([\![P]\!])$ the *underlying graph* of process P.

We say a node x occurring in a graph term is *free* if it is not bound by a restriction (νx). As shown in Fig. 3, free nodes of G are labeled with their names in the hypergraph of G, while bound nodes are not. When an edge has more than one tentacle, we usually order them clockwise, with the first one being drawn as an incoming arrow and others as outgoing arrows. If necessary, we explicitly give their order by $1,2,\ldots,k$. For an edge with only one tentacle, it is not significant whether it is shown as an incoming or outgoing edge.

[1] Actually, the union is up to their common *free nodes*, as will be defined later.

Hierarchical Graph. The graph grammar defined above only describes single CaSPiS processes that represent closed systems. However, we have to treat open systems and their compositions. For example, the graph term $[\![P|Q]\!]$ is not just $[\![P]\!]|[\![Q]\!]$, as its proper definition require some further machinery (see Fig. 5). For this we need to define a mechanism of encapsulation, called *design*, to introduce a hierarchical structure into the graphs. Like simple edges, designs need to be labeled. To this end, we assume a set $\mathcal{D}$ of *design labels* is given, and extend the graph grammar presented in Definition 2.

Definition 3 (Hierarchical graph term). *A hierarchical graph term is either a graph or a design generated by the grammar*

$$\begin{array}{ll} \textit{Graph} & G ::= \mathbf{0} \mid x \mid l(\overline{x}) \mid G|G \mid (\nu x)G \mid D\langle\overline{x}\rangle \\ \textit{Design} & D ::= L_{\overline{x}}[G] \end{array}$$

where $x \in \mathcal{N}$, $l \in \mathcal{L}$ and $L \in \mathcal{D}$.

A design $D = L_{\overline{y}}[G]$ exposes a sequence of free nodes $\overline{y}$ of its *body graph* G as its *interface nodes* (*interface* for short). For a design D, the hypergraph of $D\langle\overline{x}\rangle$ is obtained from the hypergraph of D by attaching its interface nodes to the nodes $\overline{x}$. This hypergraph is a simple *design edge*, and it has the same label as D. The hierarchical nature of hypergraphes is given by that the body graph G in a design $L_{\overline{y}}[G]$ may also contain design edges.

Recall that the hypergraphs of P and Q in Fig. 3 and their corresponding graph terms represent P and Q as closed systems. They cannot be composed. We can represent each of them, say P, as an open process by a design graph term. Instead of restricting, we expose the first control node p as the interface of the design and label it by the design label $\mathbb{P}$. Then we have the design graph terms of P and Q.

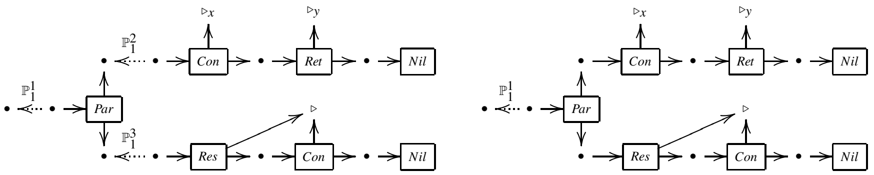
$$\begin{array}{l} [\![P]\!] = \mathbb{P}_p[(\nu p_1)(\nu p_2)(Con(p,x,p_1)|Ret(p_1,y,p_2)|Nil(p_2))] \\ [\![Q]\!] = \mathbb{P}_p[(\nu p_1)(\nu p_2)(\nu z)(Res(p,z,p_1)|Con(p_1,z,p_2)|Nil(p_2))] \end{array}$$

The hypergraphs of the two terms are re-depicted on the top and bottom of Fig. 4(a), respectively. These two hypergraphs can then be composed by linking their interfaces with an edge *Par* that also have a third node p to interface with the outside. We thus make this composed hypergraph as a design labeled by $\mathbb{P}$ and exposing p as its interface. This is the graph in Fig. 4(a), representing the parallel composition $P|Q$, and

$$[\![P|Q]\!] = \mathbb{P}_p[(\nu p_1)(\nu p_2)(Par(p,p_1,p_2)|[\![P]\!]\langle p_1\rangle|[\![Q]\!]\langle p_2\rangle)].$$

A design plays two roles in the graph representation of a process. First, it represents a service or a session as a hierarchical part of a whole process. In this case the edges[2] $\mathbb{P}_1^1$, $\mathbb{P}_1^2$ and $\mathbb{P}_1^3$ represent the designs, called abstract edges. Secondly, it represents the interface of a process through which the process communicates with its environment. When we are not interested in the hierarchical structure, a hypergraph can always be flattened by combining the nodes linked by internal abstract edges. For example, the hypergraph in Fig. 4(b) is the flat version of the hypergraph in Fig. 4(a).

[2] The use of labels such as $\mathbb{P}_1^1$ will become clarified by the formal interpretation rules of graph terms by hypergraphs.



(a) $\langle x\rangle\langle y\rangle^{\uparrow}|(\nu z)\langle z\rangle$ (b) flat version of $\langle x\rangle\langle y\rangle^{\uparrow}|(\nu z)\langle z\rangle$

Fig. 4. Graph representation with designs

3.2 Interpretation of Graph Terms by Hypergraphs

A hypergraph has different *types* of nodes for different modeling entities. In Fig. 3, for example, $\triangleright$ nodes represent data while $\bullet$ nodes represent states of the control flow. We assume a set $\mathcal{T}$ of node types and use $\mathrm{T}(x) \in \mathcal{T}$ to denote the type of node x. Each edge label or design edge label l has an arity $\mathrm{AR}(l)$ and a type $\mathrm{T}(l)$ that is the sequence of types of the nodes that the edge is associated with, thus $|\mathrm{T}(l)| = \mathrm{AR}(l)$. In our discussion, we consider only three types of nodes, i.e. $\mathcal{T} = \{\bullet, \triangleright, \diamond\}$, representing the control flow, data and channels, respectively. A design $D = L_{\overline{x}}[G]$ is *well-typed* if the sequence of its interface nodes $\overline{x}$ is of type $\mathrm{T}(L)$, while a design edge $D\langle\overline{y}\rangle$ is *well-typed* if D is well-typed and $\overline{y}$ is of type $\mathrm{T}(L)$.

To syntactically indicate whether a design edge is to be interpreted as a flattened hypergraph, we assume a designated set of design edge labels $\mathcal{F} \subseteq \mathcal{D}$, i.e. a design edge is interpreted in the flat way if and only if its label is in $\mathcal{F}$.

Definition 4 (Interpretation of graph terms). *For a graph term G, we define its interpretation by a hypergraph*

$$\mathcal{H}(G) = \langle \mathrm{N}(G),\ \mathrm{E}(G),\ \mathrm{AE}(G),\ \mathrm{fn}(G),\ \mathrm{ex}(G)\rangle$$

that is defined in Fig. 5, where $\mathrm{N}(G)$ *is the set of nodes names,* $\mathrm{E}(G)$ *the set of edges,* $\mathrm{AE}(G)$ *the set of abstract edges,* $\mathrm{fn}(G)$ *the set of free node names, and* $\mathrm{ex}(G)$ *the sequence of interface nodes.*

As discussed in the previous subsection, the interpretation of a node, edge, restriction composition of a graph term is straightforward and easy to understand. A design is generally represented by a set of binary edges, called *abstract edges*. The *source* of each abstract edge is a node in the body graph of the design which is to be exposed, and the *target* is an interface node of the design for interaction with the environment. For example, Fig. 4(a) has three abstract edges, and its flatten version Fig. 4(b) has only one, that is $\mathbb{P}_1^1$. In general, the superscript of an abstract edge label indicates which design the abstract edge represents. In Fig 4(a), we have $\mathbb{P}_1^1$, $\mathbb{P}_1^2$ and $\mathbb{P}_1^3$ that correspond to different designs. The subscript of an abstract edge label indicates which interface node of the design the abstract edge links to.

Notice that terms $(\nu x)(\nu y)G$ and $(\nu y)(\nu x)G$ are interpreted as the same hypergraph. We thus extend the restriction operator to a set of nodes and write, for example, $(\nu\{x,y\})G$ for either $(\nu x)(\nu y)G$ or $(\nu y)(\nu x)G$.

$$
\begin{aligned}
\mathcal{H}(\mathbf{0}) &\overset{\text{def}}{=} \langle\emptyset,\,\emptyset,\,\emptyset,\,\emptyset,\,\emptyset\rangle\\
\mathcal{H}(x) &\overset{\text{def}}{=} \langle\{x\},\,\emptyset,\,\emptyset,\,\{x\},\,\emptyset\rangle\\
\mathcal{H}(l(\overline{x})) &\overset{\text{def}}{=} \langle\{\overline{x}\},\,\{l(\overline{x})\},\,\emptyset,\,\{\overline{x}\},\,\emptyset\rangle\\
\mathcal{H}((\nu x)G_1) &\overset{\text{def}}{=} \langle \mathrm{N}(G_1),\,\mathrm{E}(G_1),\,\mathrm{AE}(G_1),\,\mathrm{fn}(G_1)\backslash\{x\},\,\emptyset\rangle\\
\mathcal{H}(G_1|G_2) &\overset{\text{def}}{=} \langle \mathrm{N}(G_1)\cup\mathrm{N}(G_2),\,\mathrm{E}(G_1)\cup\mathrm{E}(G_2),\,\mathrm{AE}(G_1)\cup\mathrm{AE}(G_2),\,\mathrm{fn}(G_1)\cup\mathrm{fn}(G_2),\,\emptyset\rangle\\
&\quad(\mathrm{N}(G_1)\cap\mathrm{N}(G_2)=\mathrm{fn}(G_1)\cap\mathrm{fn}(G_2))\\
\mathcal{H}(L_{\overline{y}}[G_1]) &\overset{\text{def}}{=} \langle \mathrm{N}(G_1)\cup\{\overline{y'}\},\,\mathrm{E}(G_1),\,\mathrm{AE}(G_1)\cup\{L_i^{\alpha}(\overline{y}[i],\overline{y'}[i])|1\le i\le|\overline{y}|\},\,\mathrm{fn}(G_1)\backslash\{\overline{y}\},\,\overline{y'}\rangle\\
&\quad(\overline{y'}\text{ fresh, }\mathrm{T}(\overline{y'})=\mathrm{T}(\overline{y}),\,\alpha\text{ fresh for }L)\\
\mathcal{H}(L_{\overline{y}}[G_1]\langle\overline{x}\rangle) &\overset{\text{def}}{=} \langle \mathrm{N}(G_1)[^{\overline{x}}/_{\overline{y}}],\,\mathrm{E}(G_1)[^{\overline{x}}/_{\overline{y}}],\,\mathrm{AE}(G_1)[^{\overline{x}}/_{\overline{y}}],\,(\mathrm{fn}(G_1)\backslash\{\overline{y}\})\cup\{\overline{x}\},\,\emptyset\rangle\quad(L\in\mathcal{F})\\
\mathcal{H}(L_{\overline{y}}[G_1]\langle\overline{x}\rangle) &\overset{\text{def}}{=} \langle \mathrm{N}(D)[^{\overline{x}}/\mathrm{ex}(D)],\,\mathrm{E}(D)[^{\overline{x}}/\mathrm{ex}(D)],\,\mathrm{AE}(D)[^{\overline{x}}/\mathrm{ex}(D)],\,\mathrm{fn}(D)\cup\{\overline{x}\},\,\emptyset\rangle\\
&\quad(L\notin\mathcal{F},\,D=L_{\overline{y}}[G_1])
\end{aligned}
$$

Fig. 5. Interpretation of terms

Morphism. For a formal definition of graph transformations, we need to study the relations between hypergraphs, which is captured by the notion of morphism.

Definition 5 (Morphism). *A* morphism $m\!:\!G_1\!\to\!G_2$ *is a mapping from one hypergraph G_1 to another hypergraph G_2 that satisfies the following conditions.*

1. *$m(e)$ has the same type as e, where e is either a node, an edge or an abstract edge.*
2. *If m maps an edge or abstract edge $l(\overline{x})$ to $l(\overline{y})$, m maps $\overline{x}$ to $\overline{y}$.*
3. *m maps the sequence of interface nodes of G_1 to those of G_2.*

A morphism $m\!:\!G_1\!\to\!G_2$ is *fn-preserving* if it maps each free node of G_1 to a free node of G_2 with the same node name. Two hypergraphs G_1 and G_2 are *isomorphic*, denoted as $G_1 \equiv_d G_2$, if there is a morphism between them that is bijective and fn-preserving.

Since a morphism preserves the label, source and target of abstract edges, it also preserves the hierarchical structure of hypergraphs, i.e. the layout of designs. Recall that each design is represented by a sequence of abstract edges.

3.3 Graph Transformation Rules

A graph-based theory of programming always requires the formalization of *rules of graph transformations* for defining the behavior of a program or the derivation of one program from another. A graph transformation rule is often defined in terms of the algebraic notions of *pushout*[3] [7].

Definition 6 (Double-Pushout rule). *A Double-Pushout (DPO) rule $R\!:\!GL\overset{m_l}{\leftarrow}GI\overset{m_r}{\rightarrow}GR$ is a pair of morphisms $m_l\!:\!GI\!\to\!GL$ and $m_r\!:\!GI\!\to\!GR$, where m_l is injective. Graphs GL, GI and GR are called the left-hand side, the interface and the right-hand side of the rule, respectively.*

[3] Intuitively, a pushout combines a pair of graphs with possibly some common parts by injecting them into a larger graph that is isomorphic to their disjoint union up to the common parts.

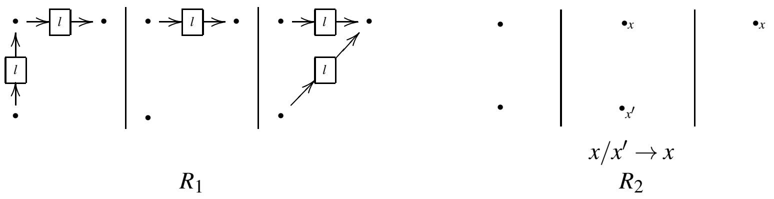


Fig. 6. Two DPO rules

$$\begin{tikzcd}
GL \arrow[d, "m_1"'] & GI \arrow[l, "m_l"'] \arrow[r, "m_r"] \arrow[d, "m_2"] & GR \arrow[d, "m_3"] \\
G & G'' \arrow[l, "m'_l"'] \arrow[r, "m'_r"] & G'
\end{tikzcd}$$

Fig. 7. A direct derivation

In many DPO rules, m_l and m_r are identity mappings or they only change a small part of the source graph. We thus simply represent a DPO rule by listing the three graphs as[4] $GL|GI|GR$ when both of its morphisms are identities, otherwise we add necessary annotations to indicate the mapping between nonidentical elements. For example, both morphisms of Rule R_1 of Fig. 6 are the identity mapping and thus no annotation is needed, but for Rule R_2, we use $x/x' \rightarrow x$ to annotate that m_r maps different nodes x and x' in the interface to the same node x in the right-hand side.

Definition 7 (Direct derivation). *Let $R: GL \overset{m_l}{\leftarrow} GI \overset{m_r}{\rightarrow} GR$ be a DPO rule. Given a graph G and a morphism $m_1 : GL \rightarrow G$, we say that G' is a* direct derivation *of G by R (based on m_1), denoted as $G \Rightarrow_R G'$, if there exist the morphisms in Fig. 7 such that m'_l, m'_r are fn-preserving and both squares are pushouts.*

In the definition, m_1 is called the *match* in the derivation as it actually matches graph GL with the subgraph $m_1(GL)$ of G. According to this match, a graph G'' is constructed by removing the elements, i.e. nodes, edges and abstract edges, in $m_1(GL \backslash m_l(GI))$ from G and preserving the elements in $m_1(m_l(GI))$. Since m_l is injective, there is a unique triple $\langle m_2, G'', m'_l \rangle$ such that m'_l is fn-preserving. Then, a graph G' is obtained from G'' by adding the elements corresponding to $GR \setminus m_r(GI)$. An example of direct derivation by Rule R_1 from Fig. 6 is shown in Fig. 8.

A graph transformation system is defined by a set δ of DPO rules, and a graph derivation is a sequential application of DPO rules of the system. Formally, G' is a *derivation* of G in system δ, denoted as $G \Rightarrow^*_\delta G'$, if there is a sequence of graphs $G_0, \ldots, G_k$ $(k \geq 0)$ such that $G \equiv_d G_0 \Rightarrow_{R_1} G_1 \Rightarrow_{R_2} \ldots \Rightarrow_{R_k} G_k \equiv_d G'$ for $R_1, \ldots, R_k \in \delta$.

[4] Here, the symbol "|" is just used to separate the graphs. It does not represent their union.

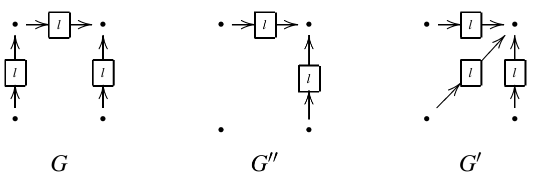


Fig. 8. An example of direct derivation by R_1

4 Graph Representation of CaSPiS

This section applies the graph algebra for representation of CaSPiS processes and uses graph transformations to study reductions of these processes. We first define a direct representation of a CaSPiS process P by a design $[\![P]\!]$. This representation is easy to understand, but would allow reductions to occur under non-static contexts. To overcome this problem, we define a *tagged version* $[\![P]\!]^\dagger$ of $[\![P]\!]$, where only tagged positions are enabled for reduction, and show that $[\![P]\!]^\dagger$ can be obtained by applying transformations rules to the untagged version $[\![P]\!]$. Then we define reductions on tagged graphs by graph transformation rules that are consistent with the process reductions.

In order to represent a process as a hierarchical graph, we define three node types $\bullet$, $\triangleright$ and $\diamond$, representing the control flow, data and channels of a process, respectively. We introduce a set of edge labels $\{Nil, Abs, Con, Ret, Sum, Par, Def, Inv, Ses, Pip, Res, rv, A\}$. Some of them represent the operators on processes, such as (Abs) for abstraction, (Par) for parallel composition and (Ses) for session. The others are for tagging (A) and restricted values (rv), which will become clear later on. The design labels represent processes ($\mathbb{P}$), service definitions ($\mathbb{D}$), invocations ($\mathbb{I}$), sessions ($\mathbb{S}$) and right-hand sides of pipelines ($\mathbb{R}$). Only design edges labeled with $\mathbb{P}$ are flat, so the hierarchy of a graph are introduced only by services, sessions and pipelines.

4.1 Processes as Designs

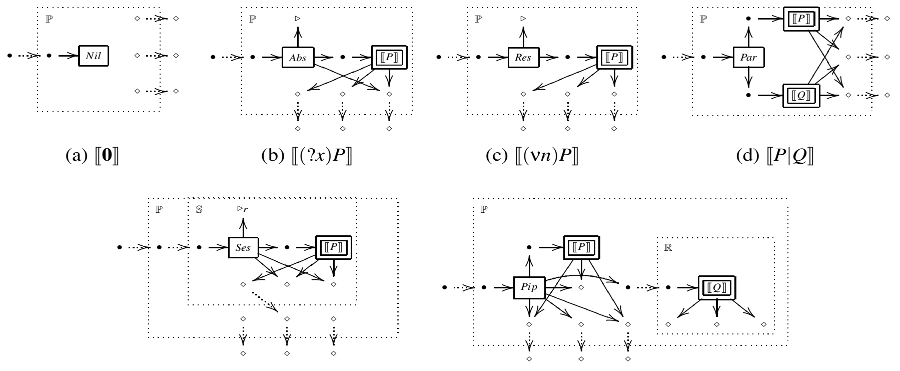
A process is represented as a $\mathbb{P}$-labeled design. The exposed nodes of such a design consist of a $\bullet$ node p, together with an input $\diamond$ node i, an output $\diamond$ node o and a return $\diamond$ node t. For a CaSPiS process P, the formal definition of its *graph representation* $[\![P]\!]$ is given in Fig. 9 by induction on the structure of P.

The nil process $\mathbf{0}$ is represented as an edge Nil (Fig. 10(a)). An abstraction $(?x)P$ is represented as a graph that attaches an edge Abs to the graph $[\![P]\!]$ and to the input channel of the whole process (Fig. 10(b)). Similar to an abstraction, a concretion and a return process is represented, but with a Con and a Ret edge associated with the output channel and the return channel, respectively. The graph of $(\nu n)P$ is composed of an edge Res and the graph $[\![P]\!]$ (Fig. 10(c)).

In the graph of a parallel composition $P|Q$ (or a sum $P+Q$), the graphs of P and Q are connected by a Par (or Sum) edge, and the channels of P and Q are combined (Fig. 10(d)). The graph of a session process $r \triangleright P$ is defined by attaching the graph of P with a session edge Ses. The Ses edge is also connected with the input and output

$$
\begin{aligned}
\llbracket \mathbf{0} \rrbracket &\overset{\text{def}}{=} \mathbb{P}_{(p,i,o,t)}[i|o|t|Nil(p)]\\
\llbracket (?x)P \rrbracket &\overset{\text{def}}{=} \mathbb{P}_{(p,i,o,t)}[(\nu\{p_1,x\})\big(Abs(p,x,p_1,i)|\llbracket P\rrbracket\langle p_1,i,o,t\rangle\big)]\\
\llbracket \langle V\rangle P \rrbracket &\overset{\text{def}}{=} \mathbb{P}_{(p,i,o,t)}[(\nu p_1)\big(Con(p,V,p_1,o)|\llbracket P\rrbracket\langle p_1,i,o,t\rangle\big)]\\
\llbracket \langle V\rangle^{\uparrow} P \rrbracket &\overset{\text{def}}{=} \mathbb{P}_{(p,i,o,t)}[(\nu p_1)\big(Ret(p,V,p_1,t)|\llbracket P\rrbracket\langle p_1,i,o,t\rangle\big)]\\
\llbracket M+M' \rrbracket &\overset{\text{def}}{=} \mathbb{P}_{(p,i,o,t)}[(\nu\{p_1,p_2\})\big(Sum(p,p_1,p_2)|\llbracket M\rrbracket\langle p_1,i,o,t\rangle|\llbracket M'\rrbracket\langle p_2,i,o,t\rangle\big)]\\
\llbracket P|Q \rrbracket &\overset{\text{def}}{=} \mathbb{P}_{(p,i,o,t)}[(\nu\{p_1,p_2\})\big(Par(p,p_1,p_2)|\llbracket P\rrbracket\langle p_1,i,o,t\rangle|\llbracket Q\rrbracket\langle p_2,i,o,t\rangle\big)]\\
\llbracket s.P \rrbracket &\overset{\text{def}}{=} \mathbb{P}_{(p,i,o,t)}[i|t|\mathbb{D}_{(p,t)}[(\nu\{p_1,i_1,o_1\})\big(Def(p,s,p_1,i_1,o_1)|\llbracket P\rrbracket\langle p_1,i_1,o_1,t\rangle\big)]\langle p,o\rangle]\\
\llbracket \bar{s}.P \rrbracket &\overset{\text{def}}{=} \mathbb{P}_{(p,i,o,t)}[i|t|\mathbb{I}_{(p,t)}[(\nu\{p_1,i_1,o_1\})\big(Inv(p,s,p_1,i_1,o_1)|\llbracket P\rrbracket\langle p_1,i_1,o_1,t\rangle\big)]\langle p,o\rangle]\\
\llbracket r\rhd P \rrbracket &\overset{\text{def}}{=} \mathbb{P}_{(p,i,o,t)}[i|t|\mathbb{S}_{(p,t)}[(\nu\{p_1,i_1,o_1\})\big(Ses(p,r,p_1,i_1,o_1)|\llbracket P\rrbracket\langle p_1,i_1,o_1,t\rangle\big)]\langle p,o\rangle]\\
\llbracket P>Q \rrbracket &\overset{\text{def}}{=} \mathbb{P}_{(p,i,o,t)}[(\nu\{p_1,p_2,o_1\})\big(Pip(p,p_1,p_2,o_1,i,o,t)|\llbracket P\rrbracket\langle p_1,i,o_1,t\rangle|\mathbb{R}_p[(\nu\{i,o,t\})\llbracket Q\rrbracket\langle p,i,o,t\rangle]\langle p_2\rangle\big)]\\
\llbracket (\nu n)P \rrbracket &\overset{\text{def}}{=} \mathbb{P}_{(p,i,o,t)}[(\nu\{p_1,n\})\big(Res(p,n,p_1)|\llbracket P\rrbracket\langle p_1,i,o,t\rangle\big)]
\end{aligned}
$$

Fig. 9. Graph representation of processes



(a) $\llbracket \mathbf{0}\rrbracket$ (b) $\llbracket (?x)P\rrbracket$ (c) $\llbracket (\nu n)P\rrbracket$ (d) $\llbracket P|Q\rrbracket$

(e) $\llbracket r\rhd P\rrbracket$ (f) $\llbracket P>Q\rrbracket$

Fig. 10. Examples of graph representation

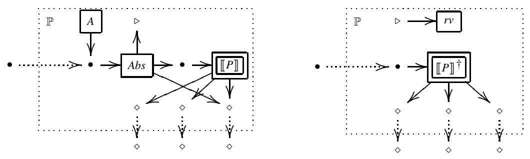
channels of P. This subgraph is enclosed in an $\mathbb{S}$-labeled design (Fig. 10(e)). The graph of a service definition or invocation is defined in the same way. A pipeline $P>Q$ is represented as an edge *Pip* connected with the graphs of P and Q, where the graph of the right-hand-side of the pipeline Q is enclosed in a $\mathbb{R}$-labeled design (Fig. 10(f)).

Notice that in Fig. 10, we enclose the body of each design by a dotted box and label it with the design label in the upper-left corner, so that the abstract edges of the design can be seen as its tentacles. We then order them clockwise, with the first one drawn as an incoming dotted arrow. In this way, all the labels of abstract edges[5] can be determined and do not need to be explicitly shown, i.e. the graph looks simpler. We will use this convention in all figures from now on. It is also worth pointing out that designs provide a natural mechanism of abstraction, enabling us to hide elements (e.g. the details of $\llbracket P\rrbracket$ or $\llbracket Q\rrbracket$ in Fig. 10) that are not significant in the current view.

[5] Recall that such labels are always complicated, with both superscripts and subscripts.

$$\begin{aligned}
[\![\mathbf{0}]\!]^\dagger &\overset{\text{def}}{=} \mathbb{P}_{(p,i,o,t)}[i|o|t|A(p)|Nil(p)]\\
[\![(?x)P]\!]^\dagger &\overset{\text{def}}{=} \mathbb{P}_{(p,i,o,t)}[(\nu\{p_1,x\})\big(A(p)|Abs(p,x,p_1,i)|[\![P]\!]\langle p_1,i,o,t\rangle\big)]\\
[\![\langle V\rangle P]\!]^\dagger &\overset{\text{def}}{=} \mathbb{P}_{(p,i,o,t)}[(\nu p_1)\big(A(p)|Con(p,V,p_1,o)|[\![P]\!]\langle p_1,i,o,t\rangle\big)]\\
[\![\langle V\rangle^\uparrow P]\!]^\dagger &\overset{\text{def}}{=} \mathbb{P}_{(p,i,o,t)}[(\nu p_1)\big(A(p)|Ret(p,V,p_1,t)|[\![P]\!]\langle p_1,i,o,t\rangle\big)]\\
[\![M+M']\!]^\dagger &\overset{\text{def}}{=} \mathbb{P}_{(p,i,o,t)}[(\nu\{p_1,p_2\})\big(A(p)|Sum(p,p_1,p_2)|[\![M]\!]\langle p_1,i,o,t\rangle|[\![M']\!]\langle p_2,i,o,t\rangle\big)]\\
[\![P|Q]\!]^\dagger &\overset{\text{def}}{=} \mathbb{P}_{(p,i,o,t)}[(\nu\{p_1,p_2\})\big(Par(p,p_1,p_2)|[\![P]\!]^\dagger\langle p_1,i,o,t\rangle|[\![Q]\!]^\dagger\langle p_2,i,o,t\rangle\big)]\\
[\![s.P]\!]^\dagger &\overset{\text{def}}{=} \mathbb{P}_{(p,i,o,t)}[i|t|A(p)|\mathbb{D}_{(p,t)}[(\nu\{p_1,i_1,o_1\})\big(Def(p,s,p_1,i_1,o_1)|[\![P]\!]\langle p_1,i_1,o_1,t\rangle\big)]\langle p,o\rangle]\\
[\![\overline{s}.P]\!]^\dagger &\overset{\text{def}}{=} \mathbb{P}_{(p,i,o,t)}[i|t|A(p)|\mathbb{I}_{(p,t)}[(\nu\{p_1,i_1,o_1\})\big(Inv(p,s,p_1,i_1,o_1)|[\![P]\!]\langle p_1,i_1,o_1,t\rangle\big)]\langle p,o\rangle]\\
[\![r\rhd P]\!]^\dagger &\overset{\text{def}}{=} \mathbb{P}_{(p,i,o,t)}[i|t|\mathbb{S}_{(p,t)}[(\nu\{p_1,i_1,o_1\})\big(Ses(p,r,p_1,i_1,o_1)|[\![P]\!]^\dagger\langle p_1,i_1,o_1,t\rangle\big)]\langle p,o\rangle]\\
[\![P>Q]\!]^\dagger &\overset{\text{def}}{=} \mathbb{P}_{(p,i,o,t)}[(\nu\{p_1,p_2,o_1\})\big(Pip(p,p_1,p_2,o_1,i,o,t)|[\![P]\!]^\dagger\langle p_1,i,o_1,t\rangle|\mathbb{R}_p[(\nu\{i,o,t\})[\![Q]\!]\langle p,i,o,t\rangle]\langle p_2\rangle\big)]\\
[\![(\nu n)P]\!]^\dagger &\overset{\text{def}}{=} \mathbb{P}_{(p,i,o,t)}[(\nu n)\big(rv(n)|[\![P]\!]^\dagger\langle p,i,o,t\rangle\big)]
\end{aligned}$$

Fig. 11. Tagged graphs of processes



$[\![(?x)P]\!]^\dagger$ $[\![(\nu n)P]\!]^\dagger$

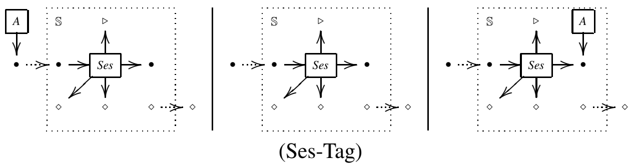
Fig. 12. Examples of tagged graphs

4.2 Tagged Graphs of Processes

In the graph term $[\![P]\!]$ of a process P, a control flow node, i.e. a $\bullet$ node, p represents the "beginning" of the graph of a sub-process Q. In this sense, *p corresponds to* a context $C[\cdot]$ with $C[Q] = P$. Recall that in a process reduction only sub-processes occurring in static contexts are allowed to interact with each other (e.g. reductions cannot take place under a prefix). Therefore to define reduction on graphs, we need to distinguish the $\bullet$ nodes, i.e. control flow nodes, that correspond to static contexts from those $\bullet$ nodes that correspond to non-static contexts. For this, we introduce unary edges labeled by A, called *tag edges*. These edges are used to tag the control flow nodes corresponding to static contexts.

Definition 8 (Tagged graph). *The tagged graph representation of P, denoted as* $[\![P]\!]^\dagger$ *is defined in Fig. 11 by induction on the structure of P.*

In a tagged graph $[\![P]\!]^\dagger$, each occurrence of abstraction, concretion, return, service definition or invocation in a static context is tagged by an A-edge. For an abstraction and a restriction, their tagged graphs are depicted in Fig. 12. Note in the case of a restriction, $[\![(\nu n)P]\!]^\dagger$ is quite different from its untagged version. In $[\![(\nu n)P]\!]^\dagger$, a new value is generated and it is denoted by an rv-labeled edge, and original Res-labeled edge in the untagged version is not needed in the tagged version any more.



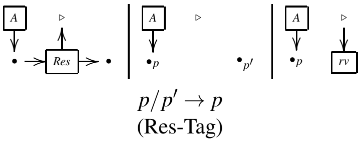


Fig. 13. Two tagging rules

It is possible to obtain a tagged graph $[\![P]\!]^\dagger$ from its untagged version $[\![P]\!]$. For this purpose, we add a tag edge to the start of the control flow of $[\![P]\!]$, and then apply a sequence of graph transformations, called *tagging rules* and denoted as Δ_T. Fig. 13 shows two interesting tagging rules. The tagging starts with a tag A at the beginning of the control. It then moves along the flow of control through a session to its body, a pipeline to its left-hand side or a parallel composition to both of its branches, before it stops at a nil process or a dynamic operator. When the tag arrives at a restriction, the restriction edge is removed, its original control nodes are combined and an *rv*-labeled edge is added.

We have proved that tagging rules are correctly defined, i.e. they indeed enable us to transform the untagged graph of any process to its tagged version.

Theorem 1 (Correctness of tagging). *For any P,* $\mathbb{P}_{(p,i,o,t)}[A(p)|[\![P]\!]\langle p,i,o,t\rangle] \Rightarrow^*_{\Delta_T} [\![P]\!]^\dagger$.

4.3 Graph Transformation Rules for Congruence

We have defined a set of graph transformation rules Δ_C to simulate the structural congruence relation between CaSPiS processes (Section 2). These rules are presented in the full version of the paper available as a technical report [6] and they include the basic rules of monoidal laws, and those for moving restrictions forward. Because of the page limit, we only give one representative rule of each kind in Fig. 14.

This set of DPO rules for congruence Δ_C is proven to be sound and complete with respect to the structural congruence of CaSPiS. The soundness means that a rule for congruence transforms the tagged graph of any process to that of a congruent process, while the completeness means that the tagged graphs of two congruent CaSPiS processes can always be transformed to the same tagged graph through application of these DPO rules. They are formalized in the following theorems.

Theorem 2 (Soundness w.r.t. congruence). *Let P be a process,* $R \in \Delta_C$ *and G be a graph. If* $[\![P]\!]^\dagger \Rightarrow_R G$*, there exist a process Q such that* $[\![Q]\!]^\dagger = G$ *and* $P \equiv_c Q$.

Theorem 3 (Completeness w.r.t. congruence). *For two processes P and Q such that* $P \equiv_c Q$*, there is a process* Q' *such that* $[\![P]\!]^\dagger \Rightarrow^*_{\Delta_C} [\![Q']\!]^\dagger$ *and* $[\![Q]\!]^\dagger \Rightarrow^*_{\Delta_C} [\![Q']\!]^\dagger$.

The proof of Theorem 2 and 3 is presented in our technical report [6].

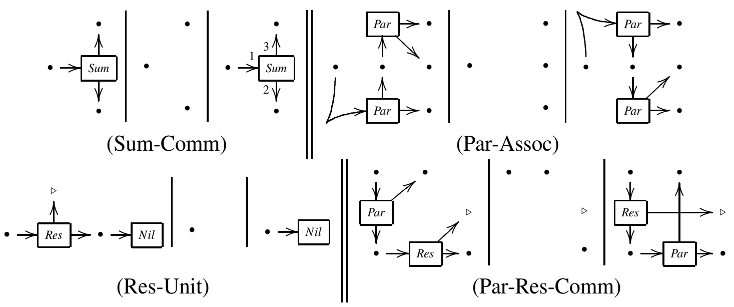


Fig. 14. Four rules for congruence

4.4 Graph Transformation Rules for Reduction

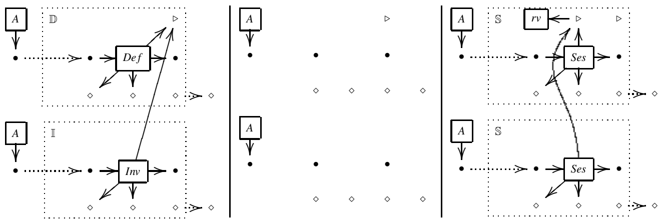
We have also defined a set of graph transformation rules Δ_R to simulate the reductions of CaSPiS processes (Section 2). We only present the two most subtle rules in Fig. 15 and explain their understanding. Rule (Ser-Sync) is for the synchronization between a pair of service definition and service invocation. The synchronization causes the creation of a new session. Note that the data node representing the service name can become isolated after the synchronization, but it can be eliminated by the garbage collection rule (D-GC) shown in Fig. 16. Rule (Ses-Sync) is for the interaction between a concretion and an abstraction on opposite sides of a session. The shared channel node by the edges *Con* and *Ses* makes sure that the concretion belongs to one of the sides. Similarly, the abstraction belongs to the other session side. Both of the abstraction and concretion are removed after the communication, with the value of the concretion assigned to variables originally bound by the abstraction.

Let Δ_A be the set of all the DPO rules provided so far, including rules for tagging, congruence and reduction. The soundness and completeness of DPO rules with respect to CaSPiS reduction is formalized below.

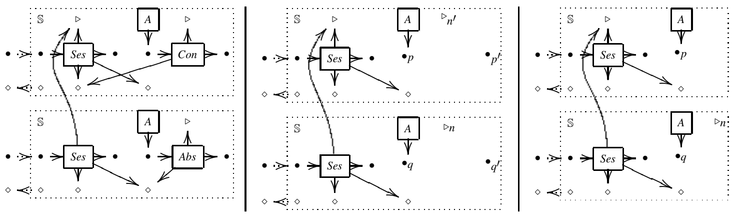
Conjecture 1 (Soundness w.r.t. reduction). For two processes P and Q such that $[\![P]\!]^\dagger \Rightarrow^*_{\Delta_A} [\![Q]\!]^\dagger$, Q can be obtained from P through a sequence of reductions, i.e. $P \longrightarrow^* Q$, where $\longrightarrow^*$ is the reflexive and transitive closure of the reduction relation $\longrightarrow$.

Conjecture 2 (Completeness w.r.t. reduction). For two processes P and Q such that $P \longrightarrow Q$, there exists a process $Q' \equiv_c Q$ such that $[\![P]\!]^\dagger \Rightarrow^*_{\Delta_A} [\![Q']\!]^\dagger$.

The proof of these conjectures is not as easy as that of Theorem 2 and 3. As for the proof of Conjecture 1, the main technical challenge is to handle intermediate states of graphs generated through applications of Δ_R. For example, after an application of (Ser-Sync), we may arrive at a state with an isolate node, and such a state does not represent any process. So, we have to show that even with various intermediate states, the graph transformation rules are still in accordance with the reduction semantics of CaSPiS. As for the proof of Conjecture 2, the main technical challenge is to deal with replications. Recall that during the reduction of a pipeline, a copy of its right-hand side is produced. However, a DPO rule consists of a fixed number of nodes and edges, thus unable to



(Ser-Sync)



$p/p' \to p,\ q/q' \to q,\ n/n' \to n$

(Ses-Sync)

Fig. 15. Two rules for reduction

(D-GC)

Fig. 16. Garbage collection of isolate data node

make a copy of a process of any size. So, it is not straightforward to define a set of DPO rules for the reductions of a pipeline that are complete. We leave the definition of such rules and the proof of Conjecture 1 and 2 as our future work.

5 An Example

Consider a service named *time* which is ready to output the current time T. This service can be used by a process that invokes the service, receives values it produces and then returns them. The composition of the service and the process is specified in CaSPiS as $P_0 = time.\langle T\rangle|\overline{time}.(?x)\langle x\rangle^{\uparrow}$. The synchronization between *time* and $\overline{time}$ creates a session with a fresh name r, and P_0 evolves to $P_1 = (\nu r)(r \rhd \langle T\rangle|r \rhd (?x)\langle x\rangle^{\uparrow})$. Then, the concretion $\langle T\rangle$ on one session side interacts with the abstraction $(?x)$ on the other side, assigning x on the latter side with T. Such an interaction makes P_1 evolve to $P_2 = (\nu r)(r \rhd \mathbf{0}|r \rhd \langle T\rangle^{\uparrow})$.

The same behavior can be simulated by graph transformations shown in Fig. 17. The first graph is the tagged graph of P_0. It is transformed to the tagged graph of P_1 by DPO rules (Ser-Sync), (D-GC) and (Ses-Tag). The tagged graph of P_1 can be further transformed to that of P_2 by Rule (Ses-Sync).

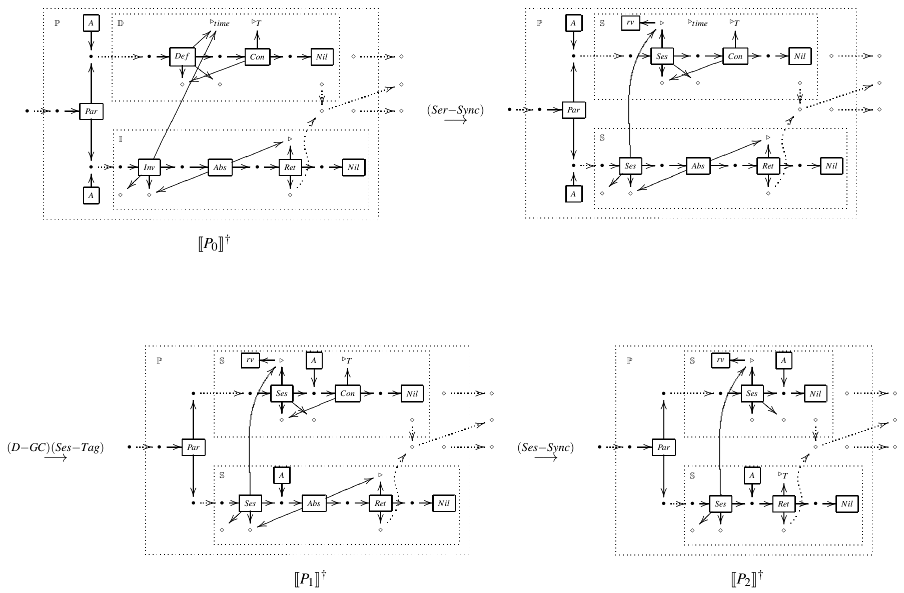


Fig. 17. Application of Graph Transformation Rules

6 Conclusion

We propose a graph characterization of structured service programming with sessions and pipelines. This is done by translating a CaSPiS process term to a design term of a graph algebra, and giving the graph algebra a model of hypergraphs. A reduction semantics of CaSPiS is then defined by a suitable graph transformation system.

The advantage of this approach is gained from the intuitive understanding of graphs[6] and the mathematical elegance and large body of theory available on graphs and graph transformations. In addition, the use of designs provides us a natural mechanism of abstraction and information hiding. This is important for the scalability of graphs. Our graph model is new compared with the one given in [5] in that hierarchy is modeled by proper combinations of abstract edges between nodes and edges of different designs. This is a key nature that enables us to define graph transformations in the DPO form.

We provide two sets of graph transformation rules, Δ_C for congruence relation between graphs (and thus for processes) and Δ_R for the reduction semantics. Δ_C is proven to be sound and complete with respect the congruence rules of CaSPiS processes. For future work, we are going to prove of soundness and completeness of Δ_R with respect to the operational semantics of CaSPiS. Although Δ_R looks not as simple as Δ_C, we are quite confident that it is sound and complete. The proof for the cases of services and sessions will be straightforward, and the only difficulty is the case of pipelines due to the dynamic creation of processes. Future work also includes the application of our graph model to a more substantial case study of service systems, and the implementation of

[6] This is more in terms of the concepts and structures of graphs than the graphic representation.

the graph model with existing graph tools. Due to the complexity of the underlying mathematical structures of graphs, we need to consider possible optimizations in the implementation so as to reduce the computation scale as well as the consumption of computer resources.

References

1. Boreale, M., Bruni, R., Caires, L., De Nicola, R., Lanese, I., Loreti, M., Martins, F., Montanari, U., Ravara, A., Sangiorgi, D., Vasconcelos, V., Zavattaro, G.: SCC: a service centered calculus. In: Bravetti, M., Núñez, M., Tennenholtz, M. (eds.) WS-FM 2006. LNCS, vol. 4184, pp. 38–57. Springer, Heidelberg (2006)
2. Boreale, M., Bruni, R., De Nicola, R., Loreti, M.: Sessions and pipelines for structured service programming. In: Barthe, G., de Boer, F.S. (eds.) FMOODS 2008. LNCS, vol. 5051, pp. 19–38. Springer, Heidelberg (2008)
3. Bruni, R.: Calculi for service oriented computing. In: Bernardo, M., Padovani, L., Zavattaro, G. (eds.) SFM 2009. LNCS, vol. 5569, pp. 1–41. Springer, Heidelberg (2009)
4. Bruni, R., Corradini, A., Montanari, U.: Modeling a service and session calculus with hierarchical graph transformation. In: Proc. of GraMoT 2010. ECEASST. EASST (2010) (to appear)
5. Bruni, R., Gadducci, F., Lluch Lafuente, A.: A graph syntax for processes and services. In: Laneve, C., Su, J. (eds.) WS-FM 2009. LNCS, vol. 6194, pp. 46–60. Springer, Heidelberg (2010)
6. Bruni, R., Liu, Z., Zhao, L.: Graph representation of sessions and pipelines for structured service programming. Technical Report 432, UNU-IIST, P.O. Box 3058, Macao (2010), `http://www.iist.unu.edu/www/docs/techreports/reports/report432.pdf`
7. Corradini, A., Montanari, U., Rossi, F., Ehrig, H., Heckel, R., Löwe, M.: Algebraic approaches to graph transformation, Part I: Basic concepts and double pushout approach. In: Rozenberg, G. (ed.) Handbook of Graph Grammars and Computing by Graph Transformation. Foundations, vol. 1, pp. 163–245. World Scientific, Singapore (1997)
8. Decker, G., Puhlmann, F., Weske, M.: Formalizing service interactions. In: Dustdar, S., Fiadeiro, J.L., Sheth, A.P. (eds.) BPM 2006. LNCS, vol. 4102, pp. 414–419. Springer, Heidelberg (2006)
9. Ferrari, G., Hirsch, D., Lanese, I., Montanari, U., Tuosto, E.: Synchronised hyperedge replacement as a model for service oriented computing. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 22–43. Springer, Heidelberg (2006)
10. Gadducci, F.: Term graph rewriting for the pi-calculus. In: Ohori, A. (ed.) APLAS 2003. LNCS, vol. 2895, pp. 37–54. Springer, Heidelberg (2003)
11. Hoare, C.A.R.: Communicating sequential processes. Comm. ACM 21(8), 666–677 (1978)
12. Jensen, O.H., Milner, R.: Bigraphs and mobile processes. Technical Report 570, Computer Laboratory, University of Cambridge (2003)
13. Lucchi, R., Mazzara, M.: A pi-calculus based semantics for WS-BPEL. Journal of Logic and Algebraic Programming 70(1), 96–118 (2007)
14. Milner, R.: Communication and Concurrency. Prentice-Hall International, Englewood Cliffs (1989)
15. Milner, R., Parrow, J., Walker, J.: A calculus of mobile processes, I and II. Information and Computation 100(1), 1–40, 41–77 (1992)
16. Misra, J., Cook, W.R.: Computation orchestration: a basis for wide-area computing. Journal of Software and Systems Modeling 6(1), 83–110 (2007)

Will the Real Service Oriented Computing Please Stand Up?

Farhad Arbab

Foundations of Software Engineering
CWI
Science Park 123
1098 XG Amsterdam
The Netherlands

1 Introduction

At the end of the FACS 2010 meeting, in the charming city of Guimarães, a panel discussion was organized around the topic of service oriented computing. The panel members were asked for their views on whether or not the rather better established field of component based software engineering subsumes service oriented computing, and if not, what exactly is the difference between component based and service oriented computing. The starting position of the majority of the panel members was that the basic concepts and formalisms at the foundation of service oriented computing are covered and investigated sufficiently well within component based software engineering. As a member of this panel, my dissenting views on this topic triggered a lively discussion that engaged the audience as well. Once the dust of the discussions arising from the usual misunderstandings of terminology settled, a near consensus became visible, and a useful distinction emerged that may lead to a better understanding of the territory and identification of directions for further research. Subsequently, a number of participants proposed to me that it may be useful to summarize this part of the discussions, and the editors of this volume agreed that such a summary complements the portrayal of FACS 2010 conveyed by the regular contributed papers in this post-proceedings. As such, this document is neither a report on nor a summary of the said panel discussion. Rather, it is a reconstruction of my personal views as presented and discussed in this panel, which does not necessarily reflect the views of others.

2 The Buzz

Indeed, service oriented computing has become very fashionable in the past few years, attracting considerable attention in both academia and industry. The flurry of activity that has produced an alphabet soup of acronyms for complementary, incompatible, and conflicting standards concerning various aspects of (Web) services has only recently ebbed. As it is usual with fashionable trends, the inevitable more sober evaluation of what exactly is the problem for which these standards and technology constitute a solution has subdued the initial over-enthusiasm depicting service oriented computing as the new panacea for all software problems. The coexistence of at least two irreconcilable worlds of Web and Grid services from the start, plus the advent of cloud computing that sapped part of that over-enthusiasm, may have speeded up this sobering somewhat. To be fair, healthy skepticism about service oriented computing has not been in short supply. Prompt proclamations of "been there, done that" were issued, although to what extent they were heeded remains anyone's guess.

If software services are supposed to offer a new paradigm for engineering of software (intensive) systems, the hard questions of "What precisely is conceptually new here?" and "What exactly are the scientific challenges, if any?" have not received the serious attention that they deserve often enough, and seldom answered satisfactorily. It is perfectly understandable for those who view this scene from outside, especially with the more austere look honed by formal methods, to conclude that beyond some hyped up standards and exchange and wrapper technology, the paradigm of software services involves nothing conceptually new or challenging, beyond the concerns of good old distributed computing and component based systems. This panel was certainly not the first time that I was confronted with such views. While there is a good deal of truth to these views, I believe that underneath the commotion, there really are serious questions and challenges that, although often neglected, must be eventually attended to for software services to both deserve recognition as constituting a new paradigm, and have an impact commensurable with their hitherto hype.

This situation reminds me of the popular TV game show To Tell The Truth, which originally ran in the 60's on CBS in the US and was syndicated in the 70's and 80's and beyond. The set up of the game involves (usually) 3 challengers and a panel of celebrities. The challengers consist of a contestant who has an unusual occupation or experience and is obligated to answer questions posed to him truthfully, accompanied by 2 impostors whose roles are to pretend to be the real contestant by giving misleading or untrue answers to questions posed to them. The panel and the audience are first presented with a truthful statement describing the unusual occupation or experience of the contestant. Then, each member of the panel of celebrities is given a fixed length of time to ask questions from each challenger, and in the end, the panel members vote to identify the contestant among the impostors. Once the votes are cast, the host asks the catch phrase question of the game show: *"Will the real [contestant] please stand up?"* The challenger and the impostors, then, reveal their true identities and the challenger wins a prize money based on the number of incorrect votes.

It seems to me that we have been confounded in identifying true service oriented computing among the challengers. As in the game show, asking the right questions based on a keen understanding of the proclaimed profile of the contestant serves us well in eliminating the impostors. Let us start to examine this profile.

3 Software Engineering

Software engineering consists of applying engineering discipline to construction of complex software intensive systems. A hallmark of all engineering disciplines is composition: Construct more complex systems by composing simpler ones. This principle allows us to derive the properties of a composed system as a composition of the properties of its constituents.

Any brief history of the evolution of software engineering must mention the milestones of structured programming with subprograms, functions, and modules, the object-oriented paradigm (OO), component-based software engineering (CBSE), and purportedly, service-oriented computing (SOC), as the contemporary tentative end of this history. SOC involves highly interactive, open, dynamic, heterogeneous, distributed applications, wherein the notion of *service* (as opposed to function, module, class, or object) constitutes the basic, atomic building block. Many examples of SOC and service oriented architectures already exist and have been considered successful in the real world. This has attracted strong academic and industrial interest, and active standardization efforts.

It is instructive to measure the milestones of this history against the hallmark of the discipline to identify trends: what composition mechanisms are afforded by each of these milestones and what properties characterize those mechanisms?

Functions and subprograms are composed using local or remote procedure calls. A procedure call in fact invokes a specific algorithm and weaves it into the algorithm of the caller. The composition mechanism in structured programming amounts to direct *composition of algorithms*. Intuitive as it may seem in the sequential world, in parallel and distributed settings, weaving algorithms by hard-coding function calls within other functions intermixes (data-dependent) computation control flow with concurrency protocols, making both more convoluted than necessary.

Method invocation (local or remote) constitutes the composition mechanism in OO. It loosens the tight coupling of algorithm weaving of function calls by introducing *object* as a level of indirection between the caller and the called algorithms. Instead of deciding on and invoking a specific algorithm in a function call, in OO the caller only identifies an *operation* to be performed by an *object*; it is (the implementer of) the target object that decides which specific algorithm to use in order to perform that operation. The caller is thus shielded from the details of the target algorithm (and the data structures that it manipulates) by the level of indirection that the target object provides, yielding a looser coupling than composition via direct function calls. The control-flow coupling between the caller and the called algorithms becomes even looser in the asynchronous method invocation mechanisms, making them even more suitable than synchronous method invocation for concurrent settings.

With few negligible exceptions, software composition via composition of functions, objects, modules, and libraries is possible only at the source language level within the same programming language. Generally, the object code of a function, class, or object produced by one programming language compiler cannot be composed with the object code of a calling program produced by another language compiler, or even another version of the same language compiler. The possibility of object-code-level composition is one of the key advantages offered by component models in CBSE. Variants of method invocation have proliferated from OO programming into the world of OO components.

Stripped of its OO method invocation semantics, asynchronous message passing constitutes a different composition mechanism based on the exchange of (locally or remotely) targeted messages. Used in concurrency platforms such as PVM and MPI, non-OO component models, and SOC, asynchronous exchange of messages through targeted-send and receive loosens the coupling between the sender and the receiver further by disallowing any control-flow coupling between the two. Targeted message passing imposes an asymmetric dependence between the sender and the receiver of a message: the sender must (know and) target the message to the receiver, but the receiver is free to receive any message from any (unknown) sender. As a composition mechanism, then, targeted message passing hinges on an asymmetric coupling: the binding of the sender to the receiver is tighter than the binding of the receiver to the sender. Untargeted message passing, as used in stream processing, dataflow, and workflow models, which are also used in SOC, provides a composition mechanism with the loosest possible coupling between the constituents that comprise a system. It binds communicating entities in symmetrically anonymous communication: neither the sender nor the receiver of a message needs to know its counterpart beforehand.

As its purported successor milestone, it is not surprising that SOC shares this same profile with CBSE. To recognize what, if anything, differentiates them, we must examine the implications of what it means to have services as the atomic building blocks in composition, and in fact, what exactly, if anything, distinguishes a service from a component.

4 Service Oriented Computing

My preferred definition of a (software) service states that *a service is the behavior that a piece of software manifests when it executes on its provider's own hardware*. Immediately, this definition distinguishes a service from a component (or function, module, class, object, library, pattern, skeleton, etc.) because it states that a service is *not* a piece of software, but the behavior that a piece of software exhibits when executed on its provider's own hardware. Viewed as *commodity behavior*, software services have less in common with pieces of software, as in components, and more in common with services that constitute the basic commodities in service economies of post-industrial societies. We are not concerned with how our mail is actually sorted and delivered. For all we know, the behavior we subscribe to as our mail service may equally well be manifested by the likes of Cliff Clavin of Cheers, or elves and gnomes: our interface with this service (pick up and delivery of mail) shields us from knowing or caring what agents manifest it or how. Similarly, for all we care to know, what lies behind the fast effective search service offered by Google may not be software at all, but armies of clever homunculi and specially trained gerbils.

Transportation services, delivery services, catering services, event organization services, janitorial services, etc. use other services and compose them with their own added value to provide new offerings in the service market. Similarly, although the building blocks of SOC involve software (intensive) systems, SOC is primarily *not* about software! SOC emphasizes the intrinsic value of the behavior of a software (intensive) system as a *business process*, with a simple client-friendly interface, that can be used directly, or in the context of composed services, by other services.

The proclaimed promise of SOC has been to transform the Web into an open global marketplace to offer electronic services. For this promise to materialize, an easy to use infrastructure is necessary to support composition of distributed services by third-parties. This fact, naturally, shows the overlap of SOC with many a concern in distributed computing. These concerns, of course, have been recognized and studied in the context of distributed components as well. However, SOC adds a new emphasis on many such concerns and presents a few challenges of its own.

Traditionally, the primary concern in distributed computing has been exploitation of resources, primarily hardware, but also other resources such as specialized databases available only on certain sites. Typically, in distributed computing one is concerned with only a single distributed application, which is virtually always closed. In this setting, composition involves pieces of software, such as components or (sub)tasks, that are tailor-made to fit and cooperate with each other, and encapsulate the hard-coded concurrency protocols necessary to do so. Heterogeneity is to be avoided as a plague, and only if that is not possible, then one may cope with it in quarantine. Autonomy of the constituents is typically a convenient pretense: a useful abstraction to achieve a reasonable structural decomposition of the application into its custom-made constituents.

The situation in SOC is different. SOC is meant to offer an open platform, not just individual closed applications. Distribution in SOC involves more than mere resources: jurisdiction and expertise are distributed as well. The expertise of the provider of a geographic location service is integral to the commodity behavior that it provides; something that is unlikely for the clients of this service to be able to duplicate cost-effectively. This provider, of course, maintains both legal jurisdiction and de facto control over the offered service. In this setting, autonomy is real and absolute. Heterogeneity is unavoidable and the norm. Composition of services involves making multiple, generally incongruent, full applications that were not necessarily designed to work with each other, do so nevertheless, without access to their source, object, or even executable code. It is unimaginable to expect service providers change the protocol of their offerings on demand to custom fit them within each new composition dreamed up by an entrepreneur who wishes to create a new composed service. The coordination and concurrency protocols for making those incongruent full applications work together must be imposed on them *exogenously*, i.e., from outside.

5 Challenges of SOC

The first technical challenge in SOC involves the aforementioned *coordinated composition from outside*. Services that are not tailor made to work with each other need some so-called glue code that resides outside of those services to make their protocols and data types compatible, and orchestrate their activities into that of a coherent whole. Naturally, such glue code can be written in any programming language. But, given the special requirements and characteristics of this type of glue code programs, can we design more suitable programming models and languages to serve this purpose?

In fact, this seems the simplest among the SOC challenges, perhaps because we understand its issues better than those of other challenges and some solutions already exist as well. Aside from conventional programming languages, scripting languages and workflow models and languages are used for this purpose, and a new language specifically intended for composition of Web services, BPEL, has also emerged, which incorporates a number of features to deal with such concepts as failure and compensation in long running transactions. Nevertheless, conceptually, the level of abstraction of BPEL is arguably, if only marginally, higher than that of, say Java or C, and using any of these alternatives to develop glue code, the simplest incongruity among to-be-composed services can require the sort of non-trivial programming that taxes the skills of the novice. The inherent concurrency of SOC makes it quite nontrivial to reason about the properties of the resulting glue code, such as conformance with the requirements, correctness, safety, liveness, etc. What the intended users in SOC can relate to consists of specifications of business processes, abstract workflows, compliance requirements, etc. Suitable models and tools to enable these users to communicate in these terms are sparse, the gap between such models and the existing glue code languages is huge, and this void spans over still missing significant concepts.

A second challenge in SOC involves search and discovery of services. Analogous to a function or class, identification of a service by its syntactic signature is trivial and standard Interface Definition Languages are utilized for this purpose. Semantic matters, however, are less trivial: recognizing differently named compatible types, types that "contain" or "overlap" with other types, and types that can be easily converted into other types, etc., requires an understanding of the "meaning" behind signature syntax. Clearly, ontologies can help here. What is far less obvious is how to describe the behavior of a service, such that a search engine can locate among the service offerings available on the Web, the suitable exact, partial, and/or close matches with the desired behavior of a required service that a user specifies.

Just as it is easy to use such tags to describe the contents of pictures, it is also easy to imagine associating ontological, structured textual, or even free text tags with every service to describe its behavior. Both schemes are unreliable (how do we know the descriptions are accurate?), impractical (who does that?), and inadequate (what ensures consistent compliance in a large population?). Advances in image processing have now placed us at the threshold of programs understanding the contents of pictures, such that, given the picture of the face of an individual, they can search for other pictures that include that individual, although his/her facial expressions, features, hair style, accessories, etc., may slightly differ in the two pictures, e.g., because the individual may have aged. Given a proper description of the behavior of a service, e.g., an automaton, how can we encode this behavior into a comprehension by programs such that we can define measures of sameness, containment, closeness, etc., for behavior in order to allow analogous searches?

A third challenge involves adaptation and decomposition of behavior (not mere syntax or ontologies). Given a proper formal specification of the behavior that a user needs and that of a behavior that contains, overlaps, or is close to it by the above measures, how can the latter be adapted into the required behavior? To do this automatically requires formal models in which we can articulate two given instances of behavior, compute their behavioral difference, and express this difference as the missing adapter that transforms one into the other. In such a setting, a provider that conjures up a new service offering can specify its desired behavior, X, play the what-if game of constructing X by some composition that incorporates the behaviors $Y_1, Y_2, ...Y_n$ of n existing service offerings as the essential ingredients of X, and automatically derive the missing behavior Z of the glue code that would be necessary to adapt and compose $X = Z \bowtie Y_1 \bowtie Y_2 ... \bowtie Y_n$. The decomposition of X using $Y = Y_1 \bowtie Y_2 ... \bowtie Y_n$ is an instance of finding the behavior Z of a missing adapter to transform Y into X.

A fourth challenge concerns a glaring reality implied by the very definition of a service as a behavior, rather than a piece of software. From functions and modules to objects and classes to components, software composition involves procuring pieces of software (as source, object, or even executable code) and integrating (perhaps slightly modified versions of) them into the medium of software that constitutes a new application. In all such constructions, procured *software touches software*. Composition of software services is different in that it involves constructions where procured service *software does not touch software.* The subtle, but important, implication of "software does not touch software" comes to light by asking "Then, what is in between?" The answer is the bane of real world applications, the dread of software people, everything that is not software: the environment!

A significant difference between software engineering and other engineering disciplines stems from the fact that the artifacts of its discourse comprise of primarily mathematical abstractions, as opposed to real world objects. Unencumbered by concerns for gravity, decay, and other forces of nature, software engineers have been free to construct any edifice imaginable, capable to last for ever, so long as it is designed and constructed correctly in the first place. Engineers of other disciplines hardly can afford the luxury of abstracting the real world out (at least, not for long).

Isolated from the forces of nature, the feeling of omnipotence in this abstract world of software artifacts induces a detrimental exaggerated sense of responsibility in software people: as the sole cause of everything a piece of software does, its developer surely must assume the responsibility for whatever that may go wrong when it runs. Perhaps this illusion of responsibility has contributed to over-indulgence with the notion of *correctness* in (at least the more formal side of) software engineering. By "over-indulgence" I do not intend to discount the importance of correctness of software, nor denigrate all the remarkable progress we have made to establish it. However, I believe "over-indulgence" is a fair description of how we have emphasized correctness to the exclusion of other relevant concerns.

Strictly speaking, by far the majority of objects, artifacts, tools, constructs, and processes (including software systems) that we use and rely on every day are incorrect; life itself is incorrect! But, none of this actually matters and we use and rely on them nevertheless, because they either work and behave correctly frequently enough, and/or we manage to bypass or work around their quirky misbehavior, most of the time. Unless and until, that is, their incorrectness or misbehavior passes our threshold of tolerance and produces a real or metaphoric disaster. Other engineering disciplines that have not enjoyed the luxury of primarily living in the abstract world of mathematical constructs immune to the forces of nature, have developed long-standing traditions that emphasize the concept of *robustness*, not instead of, but alongside correctness. There is plenty that software engineering can learn and adapt about robustness from other engineering disciplines, and SOC presents a grand theater of operation where without robustness victory will remain illusive.

Correctness and robustness are fundamentally different. Correctness presumes control over the causes of, and therefore, the ability to detect and eliminate all failure; ergo, correctness attempts to banish failure, to yield a perfectly correct artifact. Robustness presumes that it is not possible to control the causes of, detect, or eliminate all failure; ergo, robustness acknowledges and embraces the potential for failure, and instead attempts to restrict, confine, or compensate for the consequential damages that ensue from failure, when and if it happens, keeping them below the threshold of disaster.

In the setting of SOC, it seems only common sense that we should not fret over the correctness of every service; practically we cannot. If we strive to make every bit of software we construct correct, services offered by others that we use in composition with ours may not be correct. Furthermore, it still does not matter if every individual service we use, and every bit of software in between, is correct: things will still go wrong because that proverbial butterfly will still flutter its wings one sunny spring morning in a nature park in India, setting off a chain reaction that manifests a storm a few days later in Canada, that knocks down the north-east section of the power grid in the US, taking out the servers on which some of those correct services run. Corrosion and wear and tear of electrical surges still slowly take their toll on the cables, switches, and physical equipment that manifest that decay-free abstract world wherein our software rules. Robustness means we need to consider the real world and the forces of nature as the real environment in which services operate, and therefore make sure that the behavior of these services will not conspire with the failures inspired by causes beyond our control, to initiate or exacerbate undue damage.

The distributed nature of SOC adds the complications of partial failures, long running transactions, and compositional reasoning about quality of service (QoS) properties. The atomicity of multiple actions in a distributed setting cannot always be guaranteed in presence of failures, and not every action can be undone in the real world. Robustness requires at least a compensation for the effects of partially completed composite actions that cannot be rolled back. The high degree of heterogeneity and the fact that the owner of a composite service is not necessarily the owner of its building blocks, make issues involving QoS properties increasingly entangled. The end-to-end QoS of a composed service is an integral part of its robustness, often as important as its functional properties. Yet, even if the QoS properties of every individual service and glue code are known, it is far from trivial to determine and reason about the end-to end QoS of a composed system in its application context.

Of course, many of the issues involved in robustness have been studied in computer science, for instance, under the rubric of fault tolerance. But SOC adds its own new twist to robustness. Even if we assume that all conceptual issues have already been adequately addressed, it is unimaginable to expect that every provider who conjures up a new service by composing existing services, possesses the technical skills and resources to manually infuse robustness into its new offering. The software engineering challenge of robustness in SOC is to provide a distributed platform that supports the robustness of composed services, given the robustness and compensation properties of their constituent services. Preventing plug-ins to do harm in the framework of a browser is non-trivial. Unlike a browser, the framework of SOC is distributed and its intended developers of new services are far larger in numbers and far less skilled (ergo, need much more automation) than the developers of browser plug-ins.

In other engineering disciplines, control theory is used to control the behavior of dynamical systems. Although the bulk of this work deals with continuous systems, discrete and hybrid systems have attracted some attention too. In discrete dynamical systems, game theory has been used to assess the likelihood and the scope of undesirable outcomes, and to avert them by identifying the possibilities of making moves to counter or block the deleterious doings of an adversary. Providing a platform for SOC that offers automation support for robustness may entail borrowing concepts, results, and techniques from control theory and game theory, among others, and naturalizing them in this new setting.

6 Vote

My colleagues in the FACS 2010 panel initially stated that they consider SOC essentially as a repackaging of components, and were surprised by my "provocative" dissenting view. My intention, as reflected in this essay, was to show that there exists a profile of interesting issues and challenging problems distinct enough to differentiate SOC as a new phase in the evolution of software engineering. Nevertheless, sparing some preciously rare exceptions, the bulk of the activity and standards that have materialized under the rubric of SOC does not contend with the challenges in this sketched profile. As such, I must agree with the initial position of the other FACS 2010 panelists (shared by many in the audience and at large, as well) that, if the activity, not the profile, portrays SOC, then as the Dutch saying goes, SOC is *the old wine* of components *in a new bottle*. Perhaps we, especially in the academic research community, should pay more serious attention to what SOC ought to be, as the next phase in the evolution of software engineering, instead of focusing on the repackaged technologies and standards currently offered as SOC, that confirm we've already *been there, done that* with components. Addressing the challenges sketched in this essay can orient us in the right direction. Since what is currently offered as SOC does not fit its proclaimed profile closely enough to qualify as a new milestone paradigm, in this unusual round of To Tell The Truth, this panelist votes that all challengers are impostors!

Performance Verification in Complex Enterprise-Level Component Systems[⋆]

Ludwig Adam

Ludwig-Maximilians-Universität München, Germany
adaml@pst.ifi.lmu.de

1 Introduction

More and more complex enterprise-scale systems are considered to be *time-critical*: These applications provide a correct functionality only if certain time constraints for their program execution are met. In these time-critical environments, the aspect of performance becomes a relevant functional aspect and needs to be verifiable. We aim to find a solution for this problem by providing a practical approach for the implementation of complex enterprise-level systems, whose timing properties can be formally verified.

In the following we concentrate on the formal aspects of our work by giving an overview on a methodology for the formal specification of timed component behaviour and its verification in Sect. 2. We then outline our current work on using this methodology to verify execution constraints within component services in Sect. 3.

2 Output- and Input-Compatibility for TIOA and Their Verification

Pure I/O transitition systems (IOTS), commonly used for component behaviour specification, do not satisfy our requirements to specify time constraints for component behaviour. We have therefore introduced Timed I/O-Automata (TIOA) to support the specification of component behaviour, based on the original definition of Alur and Dill [3], as well as the work of Bengtson et al. [4] and, more recently, De Alfaro et al. [7]. TIOA have been covered in detail in [2], therefore we only provide an overview on the syntax and semantics of TIOA:

For a finite set Cl of *clock variables* (or *clocks*), a *clock constraint* is a boolean expression of the form *true*, *false*, or $x \bowtie n$ where $x \in Cl$ and $n \in \mathbb{N}$ with $\bowtie \in \{<, >, =, \leq, \geq\}$. A clock constraint $x \bowtie n$ is downward closed if $\bowtie \in \{<, =, \leq\}$. A *guard* is a finite conjunction over clock constraints. The set of guards over clock variables Cl is denoted by $\mathcal{G}(Cl)$. An *invariant* is a finite conjunction over downward closed clock constraints. The set of invariants over clock variables Cl is denoted by $\mathcal{I}(Cl)$. A *clock reset* for Cl is a subset of Cl. The set of clock resets is denoted by $\mathcal{R}(Cl)$.

[⋆] This work has been partially funded by the Bavarian Ministry for Economics, Infrastructure, Traffic and Technology under the IuK initiative RAJA, IUK-0805-0005.

Definition 1 (Timed I/O-Automata). *A Timed I/O- Automaton (TIOA) T is a tuple $\langle L, l^0, Cl, I, \Sigma, E\rangle$ where*

- *L is a finite set of locations,*
- *$l^0 \in L$ is the initial location,*
- *Cl is a finite set of clocks,*
- *$I : L \rightarrow \mathcal{I}(Cl)$ maps every location $l \in L$ to an invariant over Cl,*
- *$\Sigma = \Sigma^{in} \uplus \Sigma^{out} \uplus \Sigma^{int}$ is a finite set of actions, partitioned into disjoint sets of input, output and internal actions respectively, with $\tau \in \Sigma^{int}$ being the anonymous action,*
- *$E \subseteq L \times \mathcal{G}(Cl) \times \Sigma \times \mathcal{R}(Cl) \times L$ is a set of edges.*

We indicate $a \in \Sigma^{out}$ by $a!$ (postfixing a with an exclamation mark); similarly, we indicate $a \in \Sigma^{in}$ by $a?$ (postfixing a with a question mark). Moreover, when a TIOA T is given, we refer to the elements of T by using subscripts, e.g. we use L_T to refer to the set of locations of T and so on.

TIOA can be composed to networks to specify the behaviour of concurrent systems of interacting components which communicate over shared actions. In [2] we have provided an operational semantics of networks of TIOA in form of labeled IOTS following the approach of [5]. This synchronous, rendezvous-based communication of TIOA can now introduce violations of specified clock constraints. Two possible cases need to be considered: First, a TIOA trying to send an output may be blocked because there is no communication partner ready to take this output. Second, within a component specification it should be possible to specify that a certain input is required within a certain time for correct behaviour, e.g. if a component requires data from another component in a certain time frame in order to ensure its own execution constraint. For both cases we have defined a notion of compatibility: The property of *timed output-compatibility* ensures, that for every state and time in which a component tries to send an output, this output is immediately accepted by a communication partner. An input requirement for TIOA can be specified using explicit time-locks with location invariants [2,6]. If such an input requirement is encountered, the property of *input-compatibility* ensures that a correct input is received and is received in time. Using syntactic transformation rules for TIOA both notions of compatibility can be verified in model-checkers for Timed Automata [2], e.g. the UPPAAL model-checker [1].

3 Services in Component Behaviour

When we impose time constraints a component not only acts as a entity in the system that reacts on some input but can be rather seen as an entity that performs a set of specific tasks or services in a given time.[1] With this perspective we gain a different understanding of component behaviour: A component executes one or more services that are provided on its ports. The component behaviour

[1] We will use the notion of service not in the meaning of entities of a SOA but rather in its original meaning of a function that is provided by an entity within the system - in our case the components.

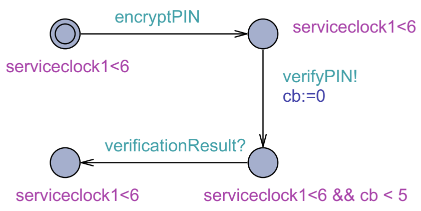
specification acts as a kind of service orchestration. Ports not only act as well-defined communication endpoints between components but they also act as service endpoints for services provided by the component: Each provided operation corresponds to the *initiation of a service* to be executed by the component and a call to a required operation can be understood as a *service request*. Execution constraints are then specified for service executions rather for the whole component behaviour. Based on their external behaviour we classify these services into *dependent* and *independent* services: While independent services do not communicate with other components during their program execution, dependent services require some external communication for correct behaviour. Dependent services can be categorized further into services with or without callbacks. In our current work we are now interested in the verification of time constraints within service executions. Especially we want to focus on possible violations of clock constraints within dependent services that are caused by wrong behaviour of the communication partner. This can be achieved by specifying single service executions as TIOA that follow certain well-formedness criteria.

Definition 2 (Well-Formedness). *Let T be a TIOA. T is a well-formed service execution specification, if following criteria are met:*

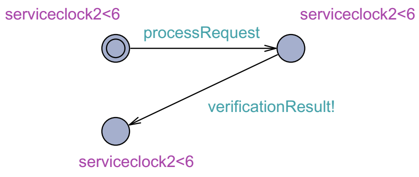
- *T has a local service clock $\mathsf{serviceclock}$ that is reset with the service initiation. Given a service constraint value sc for the overall execution of the service, every location of T has at least the service invariant $serviceclock < sc$.*
- *No mixed locations are allowed within T, i.e. there exists no location $l \in L_T$ and no $a \in \Sigma_T^{in}$ and no $b \in \Sigma_T^{out} \cup \Sigma_T^{int}$ with $(l, a, g, r, l') \in E_T$ and $(l, b, g, r, l') \in E_T$.*
- *Every input in T is specified as a input requirement as defined in [2], i.e. if there exists $a \in \Sigma_T^{in}$ and $(l, a, g, r, l') \in E_T$, then there exists a clock c and a constraint value $\mathsf{deadline}$, s.t. $I_T(l) = c < deadline$.*

Given two dependent services S_1 and S_2 that are represented by two well-formed service execution specifications T_1 and T_2 we can now use the properties of output-compatibility and input-compatibility to check for potential violations of service constraints: If S_1 is a dependent service without callbacks we can use the verification of output-compatibility to check if the outputs of S_1 are immediately accepted by S_2. If S_1 is a dependent service with callbacks we can use the verification of input-compatibility to check if the callback is sent in time by S_2. The other direction works the same.

Example 1. Fig. 1 shows an example of two service execution specifications for an ATM service that calls another service for its PIN verification. Both services are specified to terminate within 6 time units. However, the ATM service requires, that the PIN verification is done within 5 ticks for correct behaviour. This is clearly violated by the corresponding service as the verification-result may be available only after more than 5 ticks. Verification of timed input-compatibility of these two automata shows that these automata are indeed not compatible.



(a) ATM service



(b) Service execution specification for verifyPIN

Fig. 1. Example service execution specifications

4 Conclusion and Future Work

We have outlined how execution constraints can be specified and verified for service specifications for components. While we are confident that timed output- and input-compatibility is a sound method to check for violations of service constraints we still need to provide a formal proof. This includes the formal definiton of service declarations and service execution specifications as outlined in Sect. 3. Additionally, in order to provide a complete framework approach we will need to define a refinement relation to show that the component implementations are correct with regards to the specified services. Currently we are also evaluating different possibilities to specify the full behaviour of a component given a set of service specifications, as TIOA do not seem to be the ideal choice for this.

References

1. UPPAAL Model Checker, `http://www.uppaal.com/`
2. Adam, L.: Verification of timed output-compatibility and timed input-compatibility in networks of timed input/output-automata. Submitted for ECRTS 2011 (2010)
3. Alur, R., Dill, D.: A theory of timed automata. In: Huizing, C., de Bakker, J.W., Rozenberg, G., de Roever, W.-P. (eds.) REX 1991. LNCS, vol. 600, pp. 45–73. Springer, Heidelberg (1992)
4. Bengtsson, J., Larsen, K.G., Larsson, F., Pettersson, P., Yi, W.: Uppaal — a Tool Suite for Automatic Verification of Real–Time Systems. In: Alur, R., Sontag, E.D., Henzinger, T.A. (eds.) HS 1995 Part III. LNCS, vol. 1066, pp. 232–243. Springer, Heidelberg (1996)
5. Bengtsson, J., Yi, W.: Timed automata: Semantics, algorithms and tools. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) Lectures on Concurrency and Petri Nets. LNCS, vol. 3098, pp. 87–124. Springer, Heidelberg (2004)
6. Bowman, H.: Modelling timeouts without timelocks. In: Katoen, J.-P. (ed.) ARTS 1999. LNCS, vol. 1601, pp. 334–353. Springer, Heidelberg (1999)
7. de Alfaro, L., Henzinger, T.A., Stoelinga, M.: Timed interfaces. In: Sangiovanni-Vincentelli, A.L., Sifakis, J. (eds.) EMSOFT 2002. LNCS, vol. 2491, pp. 108–122. Springer, Heidelberg (2002)

Runtime Programming through Model-Preserving, Scalable Runtime Patches

Christoph M. Kirsch[1], Luís Lopes[2], Eduardo R.B. Marques[2], and Ana Sokolova[1]

[1] Department of Computer Sciences, University of Salzburg
[2] CRACS/INESC-Porto LA, Faculdade de Ciências, Universidade do Porto

1 Introduction

We propose a methodology for flexible software design, runtime programming, by means of incremental software modifications at runtime. Runtime programming acknowledges that software designs are often incomplete, and require the flexibility of change, e.g., fixing bugs or introducing new features, without disruption of their service. This flexibility is much needed for critical software that generally needs to handle uncertainty, e.g. cloud computing or cyber-physical systems, due to dynamic requirements of composition, service, or performance. Runtime modifications should be allowed recurrently, and, thus, be handled as a common case of system functionality in predictable and efficient manner, with proper understanding of inherent functional and non-functional aspects. The work in many diverse research communities with related concerns typically tends to take a partial and domain-specific view of the problem, hence comprehensive and general methodologies are in order.

In this extended abstract, we make a summary of the runtime programming proposal of [4]. The work follows up on a preliminary formulation of runtime programming [3], and work on modular compilation of real-time programs [2].

2 Runtime Programming through Runtime Patches

The runtime programming abstraction is illustrated in Fig. 1. A program (bottom) is subject at runtime to recurrent incremental modifications, called runtime patches, by an external program, a runtime patcher (top). A runtime patch determines a switch between two program specifications and states of these programs, by replacing a component in the source program. Runtime patches are applied by the patcher in congruence with program state and the (evolving) program does not stop, instead it flows with

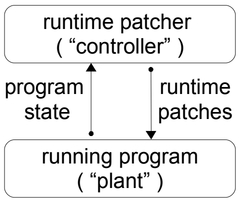


Fig. 1. Runtime programming [4]

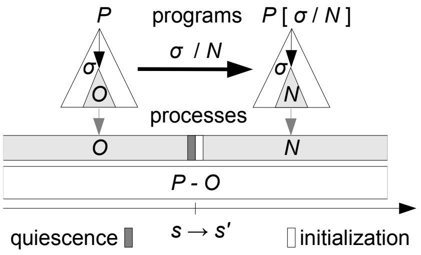


Fig. 2. A runtime patch [4]

any introduced runtime patches. An obvious analogy exists with the "controller-plant" formulation of control theory: the evolving program is the "plant", the patcher is the "controller", and runtime patches define the "control".

The transformation defined by a runtime patch, illustrated in Fig. 2, has well-defined syntactic and semantic effects.

Syntactically, a patch σ/N over program P defines the substitution of a component at "program path" σ, $O = P[\sigma]$, by component N, yielding program $P\ [\sigma/N]$. Strict component addition or removals are special instances of this effect, when O or N are undefined respectively.

Semantically, the patch defines an instantaneous switch from a state s of P to a state s' of $P\ [\sigma/N]$ such that: the processes of O terminate, according to a notion of "quiescence" in place that expresses graceful conditions for doing so (e.g., inactivity, or atomic instants that marks the end of a component's "transaction"); the processes of N start in a valid initial state; and the state of processes of other components $P - O$ is unaffected ($s[P - O] = s'[P - O]$).

Thus, runtime programming assumes only a simple abstraction of component-based software, comprised of a modular relation between (the syntax of) components and (the semantics of) processes, plus built-in notions of initialization and quiescence.

3 Model Preservation

We consider that runtime programming should be model-preserving. i.e., preserve the programming model in place for programs in terms of program syntax, semantics, and correctness properties. More precisely, model preservation is the guarantee that, in a runtime programming system, a proper program is running at all times, and a corresponding state for that program is observed that complies with correct operation. The point is avoiding an "ad-hoc", disruptive nature for runtime patches, and relying on no particular abstraction level other than the one already in place for programs.

The concept of a runtime patch, described above, already provides relatively strong model-preserving provisions. A runtime patch defines an atomic switch, and observes proper quiescence, initialization, and isolation of replaced, new, and unchanged components, respectively. This means that in a runtime programming system, a proper program is running at all times, and a corresponding state of that program is observed, with a safe continuous flow observed upon patch effect. Hence there are no transient "meta-programs" or "meta-behavior" that are alien to the syntax or semantics of programs.

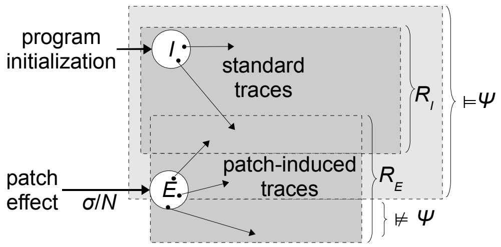


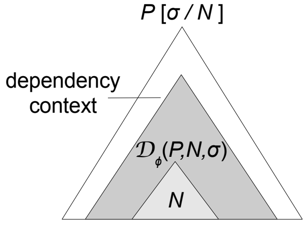
Fig. 3. Model preservation [4]

Additionally, we should expect a program that is "started" through patch effect behaves correctly. That is, its execution should conform to any particular properties of correctness for the program, were it to execute from scratch. The problem, however, illustrated in Fig. 3, is that a runtime patch σ/N over a given program P defines a partial, "live" initialization of $P\ [\sigma/N]$, rather than a whole-program initialization. So it could happen that "patch-induced traces", those with states R_E resulting from effect E of σ/N over the flow of P, do not conform to a certain expected property ψ of correctness that holds for "standard traces" of $P\ [\sigma/N]$, those with states R_I resulting from overall initial conditions I. We say the patch is model-preserving if and only if R_E is a subset of the satisfiability state space of the correctness property ψ. Checking model preservation should be an integral part of the process of patch compilation, described next.

4 Scalability

The complexity of a runtime programming system should ideally scale with the "size" of runtime patches. Such complexity comes from patch compilation, the set of procedures required to verify and integrate a patch, such as checking a model-preserving nature for patches, and other aspects like code generation or re-linking. If patch compilation does not scale in the general case, the practicality of runtime programming is compromised.

To characterize scalability of patch compilation, we propose the incremental compilation framework illustrated in Fig. 4. The idea is that for a patch σ/N over



$$I(\phi) = (\psi_\phi, \mathcal{D}_\phi, \mathcal{C}_\phi)$$

$$\frac{P \models \phi \quad \mathcal{D}_\phi(P, N, \sigma) \models \psi_\phi}{P\ [\sigma/N] \models \phi}$$

$$\mathcal{C}_\phi(P, N, \sigma) = \mathrm{O}\left(\mathcal{D}_\phi(P, N, \sigma) \models \psi_\phi\right)$$

Fig. 4. Scalability [4]

program P, compilation should be incremental to that of P, and scale appropriately in proportion to the extent of the patch, as determined by P, σ and N. For each aspect of compilation ϕ (e.g. code generation), an incremental strategy should be defined, $I(\phi) = (\psi_\phi, \mathcal{D}_\phi, \mathcal{C}_\phi)$ with the following rationale. For a patch σ/N over (the previously compiled) P, ϕ should be dealt with for $P\,[\sigma/N]$ by some incremental effort ψ_ϕ over a dependency context of components $\mathcal{D}_\phi(P, N, \sigma)$. The complexity of checking ψ_ϕ over $\mathcal{D}_\phi(P, N, \sigma)$ by some algorithm is expressed by $\mathcal{C}_\phi(P, N, \sigma) = \mathrm{O}\,(\mathcal{D}_\phi(P, N, \sigma) \models \psi_\phi)$, which we call the compilation cost. In Fig. 4, it is shown (left) that the dependency context $\mathcal{D}_\phi(P, N, \sigma)$ is a set of components within $P\,[\sigma/N]$, and, additionally, it may also include the "old" component $O = P\,[\sigma]$. The inference rule on the right of the figure expresses the incrementality in compilation: under the assumption that P already verifies ϕ, it is just required to verify ψ_ϕ over $\mathcal{D}_\phi(P, N, \sigma)$.

This formulation inherently characterizes incremental compilation and its scalability, in the size (dependency context) and time (compilation cost) dimensions. Scalability can be broken in one of the dimensions, e.g., when a patch requires the full program as context, or if the incremental effort is intractable, as measured by the compilation cost. A good degree of scalability corresponds to a small dependency context, and a tractable incremental effort. To achieve it, the choice of compilation strategy may in some cases represent a loss of precision. A strategy that scales well, and covers the more general cases of valid patches reasonably, will be preferable to one that is more exact, but scales poorly. This is important in particular when we are faced with the well-known "state explosion problem" incurred by an exact analysis.

5 Ongoing Work

In [4] we provide a detailed description and formalization of runtime programming, corresponding to the general overview given here. Additionally, we put the formulation in perspective with a case-study instantiation of runtime programming for a component-based language for distributed real-time systems, the Hierarchical Timing Language (HTL) [2,1]. In earlier work [3], some of these ideas and HTL runtime patching were discussed in preliminary short form but mainly considering the specific context of real-time systems and HTL. An incremental compilation framework was proposed for HTL already in [2], which we generalized now for component-based systems in the context of runtime programming.

Acknowledgements. This work has been supported by the EU ArtistDesign Network of Excellence on Embedded Systems Design, the Austrian Science Funds P18913-N15 and V00125, and Fundação para a Ciência e Tecnologia funds PTDC/EIA/71462/2006 and SFRH/BD/29461/2006.

References

1. Ghosal, A., Henzinger, T., Iercan, D., Kirsch, C., Sangiovanni-Vincentelli, A.: A hierarchical coordination language for interacting real-time tasks. In: Proc. International Conference on Embedded Software (EMSOFT), pp. 132–141. ACM, New York (2006)

2. Henzinger, T., Kirsch, C., Marques, E., Sokolova, A.: Distributed, modular HTL. In: Proc. Real-Time Systems Symposium (RTSS), pp. 171–180. IEEE, Los Alamitos (2009)
3. Kirsch, C., Lopes, L., Marques, E.: Semantics-Preserving and Incremental Runtime Patching of Real-Time Programs. In: Online Proc. Workshop on Adaptive and Reconfigurable Embedded Systems (APRES). pp. 3–7. ARTIST Network of Excellence (2008)
4. Kirsch, C., Lopes, L., Marques, E., Sokolova, A.: Runtime Programming through Model-Preserving, Scalable Runtime Patches. Tech. Rep. 2010-08, Department of Computer Sciences, University of Salzburg (2010)

Steps on the Road to Component Evolvability[⋆]

Mario Bravetti[1], Cinzia Di Giusto[2], Jorge A. Pérez[3], and Gianluigi Zavattaro[1]

[1] Laboratory FOCUS (Università di Bologna / INRIA)
[2] INRIA Grenoble - Rhône-Alpes
[3] CITI - Department of Computer Science, FCT New University of Lisbon

Abstract. We have recently developed a calculus for *dynamically evolvable* aggregations of components. The calculus extends CCS with primitives for describing components and their evolvability capabilities. Central to these novel primitives is a restricted form of *higher-order communication* of processes involved in update operations. The origins of our calculus for components can indeed be traced back to our own previous work on expressiveness and decidability results for *core* higher-order process calculi. Here we overview these previous works, and discuss the motivations and design decisions that led us from higher-order process calculi to calculi for component evolvability.

Introduction. The deployment of applications by the *aggregation* of elementary blocks (modules, components, Web services, ...) is a long-standing principle in software engineering. Our interest is in the *correctness* of aggregations of *components* which are subject to *evolvability* and *adaptation* concerns. The term "component" is used here in a broad sense, as it refers to elementary blocks such as Web services in cloud computing scenarios, but also to analogous concepts in different settings, such as services in service-oriented computing or long-running processes in workflow management.

To this end, we have recently defined $\mathcal{E}$, a process calculus equipped with primitives for describing components and their evolvability. Using $\mathcal{E}$ as a basis, we have studied the decidability of verification problems associated to the correctness of aggregations of components [2]. In this short paper, we present $\mathcal{E}$ and discuss the origins and motivations that led to its definition. In particular, we elaborate on the relationship between the notion of component evolvability in $\mathcal{E}$ and *higher-order* process calculi.

Steps towards Specification Languages. *Higher-order process calculi* are calculi in which processes can be passed around in communications. Higher-order (or *process-passing*) concurrency is often presented as an alternative paradigm to the first-order (or *name-passing*) concurrency of the π-calculus [8] for the description of mobile systems. As in the λ-calculus, higher-order process calculi involve *term instantiation*: a computational step results in the instantiation of a variable with a term, which is copied as many times as there are occurrences of the variable. The basic operators of these calculi are usually those of CCS [7]: parallel composition, input and output prefix, and restriction. Replication and recursion can be encoded. Proposals of higher-order process calculi include the higher-order π-calculus [10], Homer [5], and Kell [11].

[⋆] Supported by the EU integrated project HATS, the Fondation de Coopération Scientifique Digiteo Triangle de la Physique, and FCT / MCTES (CMU-PT/NGN44-2009-12) - INTERFACES.

With the purpose of investigating expressiveness and decidability issues in the higher-order paradigm, a *core* higher-order process calculus, called HOCORE, was introduced [6]. HOCORE is *minimal*, in that only the operators strictly necessary to obtain higher-order communications are retained. Most notably, HOCORE has no restriction operator. Thus all names are global, and dynamic creation of new names is impossible. The grammar of HOCORE processes is:

$$P ::= a(x).\, P \;\Big|\; \overline{a}\langle P\rangle \;\Big|\; P \parallel P \;\Big|\; x \;\Big|\; \mathbf{0}$$

An input process $a(x).\, P$ can receive on name a a process to be substituted in the place of x in the body P; an output message $\overline{a}\langle P\rangle$ sends the output object P on a; parallel composition allows processes to interact. As in CCS, in HOCORE processes evolve from the interaction of complementary actions; this way, e.g., $\overline{a}\langle P\rangle \parallel a(x).\, Q \longrightarrow Q\{^{P}\!/x\}$ is a sample reduction. See [6,9] for a complete account on the basic theory of HOCORE.

While considerably expressive, HOCORE is far from a specification language for settings involving (forms of) higher-order communication. For instance, it lacks primitives for describing the *localities* into which distributed systems are typically abstracted. Similarly, HOCORE also lacks constructs for influencing the execution of a running (higher-order) process. This is a particularly sensible requirement for the specification of systems featuring forms of evolvability and/or dynamic reconfiguration. In order to deal with those aspects, higher-order process calculi such as Homer and Kell provide primitives that allow to *suspend* running processes. In a nutshell, such primitives rely on named *localities* in which processes can execute and interact with their environment, but also in which their execution can be stopped at any time by interaction with complementary input actions. This way, the suspension of a running process is assimilated to regular process communication. Let us illustrate these intuitions by considering the extension of HOCORE with process suspension. Let $a[P]$ denote the process P inside the so-called *suspension unit* a. Assuming a labelled transition system (LTS) with actions of the form $P \xrightarrow{\alpha} P'$, process suspension is formalized by the following two rules:

$$[\textsc{Trans}]\;\; P \xrightarrow{\alpha} P' \Rightarrow a[P] \xrightarrow{\alpha} a[P'] \qquad\qquad [\textsc{Susp}]\;\; a[P] \xrightarrow{a\langle P\rangle} \mathbf{0}$$

where $a\langle P\rangle$ corresponds to the output action in the LTS of HOCORE (see [6]). As a simple example, process $S = a[P_1] \parallel a(x).\, b[x \parallel x]$ defines a process P_1 running at locality a, in parallel with an input action which may suspend the content of a and relocate two copies of it into locality b. Assuming that P_1 evolves into P_2, and given the above two rules, a possible evolution for S is the process $b[P_2 \parallel P_2]$. Observe how term instantiation plays a prominent rôle in mechanisms for process suspension.

In spite of this simple formulation, we observe that suspension primitives are not entirely satisfactory for describing evolvability as in component systems. The reason is that by assimilating suspension to communication, the evolvability of a running process is *decoupled* into two phases: (i) one in which the state of the process is actually suspended and captured and (ii) one in which the suspended process state is used within a new context. In the previous example: the first phase corresponds to capturing the state at a as P_2, while the second corresponds to substituting P_2 twice inside locality b. By considering that update actions are typically atomic operations in which suspension and relocation occur at the same time, this decoupling turns out to be not realistic in terms of modeling purposes.

Given the above considerations, in [2] we have defined *Evolvable CCS* (abbreviated $\mathcal{E}$), a variant of CCS without restriction and relabeling, and extended with primitives that allow for process evolvability in a "coupled" style. As in CCS, in $\mathcal{E}$ processes can perform actions or synchronize on them. The grammar of $\mathcal{E}$ extends CCS with *update prefixes* and a primitive notion of *component*, denoted $a[P]$:

$$P ::= \pi.\,P \;\Big|\; a[P] \;\Big|\; P \parallel P \;\Big|\; !\pi.\,P \;\Big|\; \mathbf{0} \qquad \pi ::= a \;\Big|\; \overline{a} \;\Big|\; \widetilde{a}\{U\}$$

where the U in the update prefix $\widetilde{a}\{U\}$ represents a context, i.e., a process with some holes $\bullet$. We use a and $\overline{a}$ to denote atomic input and output actions, respectively. The rest of the syntax follows standard lines. Evolution at a is realized by the interaction of component $a[P]$ with the update action $\widetilde{a}\{U\}$, which leads to process $U\{P/\bullet\}$, i.e., the process obtained by replacing every hole in U by P. The previous example would be written in $\mathcal{E}$ as the process $S' = a[P_1] \parallel \widetilde{a}\{b[\bullet \parallel \bullet]\}$, which evolves to $b[P_2 \parallel P_2]$ in a single reduction. This way, evolvability relies on the term instantiation feature of higher-order languages in a more disciplined way, ensuring atomicity in updates.

Steps towards Decidability of Verification Problems. We are interested in two correctness properties for $\mathcal{E}$ processes. The first one, *k-bounded adaptation* (abbreviated $\mathcal{BA}$) ensures that, given a finite k, at most k errors can arise during the system evolution. The second property, *eventual adaptation* (abbreviated $\mathcal{EA}$), is similar but weaker: it ensures that the system will eventually reach a state from which no other error will arise (that is, only finitely many errors can occur). Both these properties are undecidable for $\mathcal{E}$ processes, as we have shown that $\mathcal{E}$ is a Turing complete model (see [2]). The challenge is then to identify fragments of $\mathcal{E}$ expressive enough so as to represent useful evolvability patterns and for which $\mathcal{BA}$ and/or $\mathcal{EA}$ are still decidable.

A similar scenario was addressed in [3,9] for the case of HOCORE. In spite of its minimality, HOCORE was shown to be Turing complete [6]. As studied in [3,9], central to the expressiveness of HOCORE is the ability of *forwarding* a received process within an *arbitrary context.* We then investigated $\text{HO}^{-\mathsf{f}}$, a fragment of HOCORE in which the kind of processes that can be communicated is limited: in $\text{HO}^{-\mathsf{f}}$, output objects can only correspond to the parallel composition of statically known closed processes (i.e., without free variables) with processes received in previously executed input actions. This limitation to forwarding proved to be effective in terms of verification, as termination for $\text{HO}^{-\mathsf{f}}$ processes was shown to be decidable. From a pragmatic perspective, $\text{HO}^{-\mathsf{f}}$ is able to abstract those scenarios in which objects can be passed around and can only be modified by "appending" to them new objects that admit no inspection. This is the case of, e.g., the mobility of already compiled code, on which it is not possible to apply inverse translations (such as, e.g., reverse engineering).

The study in [3,9] thus suggests that key to the decidability of verification problems for higher-order process calculi is the kind of contexts allowed in higher-order actions. In turn, this is closely related to the term instantiation feature discussed before. Based on this observation, we considered constraints on the ways in which components can be updated in $\mathcal{E}$. As a result, we obtained six variants of $\mathcal{E}$ via two orthogonal characterizations. The first characterization is *structural*, and distinguishes between *static* and *dynamic* topologies of component aggregations. In a static topology the number of components does not vary along the evolution of the system: components cannot be destroyed nor new components can appear. In contrast, this restriction is not considered in dynamic topologies. The second characterization is *behavioral*, and concerns *update patterns* (i.e., the context U in $\widetilde{a}\{U\}$) which define the behavior of components after an update action. We identified three update patterns, which determine three families of $\mathcal{E}$ calculi—denoted $\mathcal{E}^1$, $\mathcal{E}^2$, and $\mathcal{E}^3$, respectively. The first update pattern admits all kinds of contexts, and so it represents the most expressive form of update. In particular, holes $\bullet$ can appear behind prefixes. The second update pattern forbids such guarded holes in contexts. In the third update pattern we additionally require contexts to have exactly one hole, thus preserving the existing behavior (and possibly adding new behaviors): this is the most restrictive form of update that we consider. These variants of $\mathcal{E}$ capture a fairly ample spectrum of scenarios. They borrow inspiration from existing component models, development frameworks, and programming languages. In [2], we have obtained the following (un)decidability results for $\mathcal{BA}$ and $\mathcal{EA}$ in the different variants of $\mathcal{E}$:

	Dynamic Topology	Static Topology
$\mathcal{E}^1$	$\mathcal{BA}$ undec / $\mathcal{EA}$ undec	$\mathcal{BA}$ undec / $\mathcal{EA}$ undec
$\mathcal{E}^2$	$\mathcal{BA}$ dec / $\mathcal{EA}$ undec	$\mathcal{BA}$ dec / $\mathcal{EA}$ undec
$\mathcal{E}^3$	$\mathcal{BA}$ dec / $\mathcal{EA}$ undec	$\mathcal{BA}$ dec / $\mathcal{EA}$ dec

The decidability of $\mathcal{EA}$ is shown by resorting to Petri nets, while for $\mathcal{BA}$ we consider results for the theory of *well-structured transition systems* [4,1]. The undecidability results are obtained by resorting to termination problems in Turing complete models.

References

1. Abdulla, P.A., Cerans, K., Jonsson, B., Tsay, Y.-K.: Algorithmic analysis of programs with well quasi-ordered domains. Inf. Comput. 160(1-2), 109–127 (2000)
2. Bravetti, M., Giusto, C.D., Pérez, J.A., Zavattaro, G.: Adaptable Processes. Technical report, University of Bologna (2011), Draft in, `http://www.japerez.phipages.com`
3. Di Giusto, C., Pérez, J.A., Zavattaro, G.: On the expressiveness of forwarding in higher-order communication. In: Leucker, M., Morgan, C. (eds.) ICTAC 2009. LNCS, vol. 5684, pp. 155–169. Springer, Heidelberg (2009)
4. Finkel, A., Schnoebelen, P.: Well-structured transition systems everywhere! Theor. Comput. Sci. 256(1-2), 63–92 (2001)
5. Hildebrandt, T., Godskesen, J.C., Bundgaard, M.: Bisimulation congruences for homer — a calculus of higher order mobile embedded resources. Technical Report TR-2004-52, IT University of Copenhagen (2004)
6. Lanese, I., Pérez, J.A., Sangiorgi, D., Schmitt, A.: On the expressiveness and decidability of higher-order process calculi. Inf. Comput. 209(2), 198–226 (2011)
7. Milner, R.: Comunication and Concurrency. Prentice-Hall, Englewood Cliffs (1989)

8. Milner, R., Parrow, J., Walker, D.: A Calculus of Mobile Processes, I. Inf. Comput. 100(1), 1–40 (1992)
9. Pérez, J.A.: Higher-Order Concurrency: Expressiveness and Decidability Results. PhD thesis, University of Bologna (2010), Draft in, `http://www.japerez.phipages.com`
10. Sangiorgi, D.: Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms. PhD thesis CST–99–93, University of Edinburgh, Dept. of Comp. Sci. (1992)
11. Schmitt, A., Stefani, J.-B.: The kell calculus: A family of higher-order distributed process calculi. In: Priami, C., Quaglia, P. (eds.) GC 2004. LNCS, vol. 3267, pp. 146–178. Springer, Heidelberg (2005)

Towards Linear Algebras of Components

Hugo Daniel Macedo[1] and José Nuno Oliveira[2]

[1] MAPi Doctoral Programme, Portugal
hmacedo@di.uminho.pt
[2] Minho University, Portugal
jno@di.uminho.pt

1 Introduction

In a recent article [1], David Parnas questions the traditional use of formal methods in software development, which he considers an underdeveloped body of knowledge and therefore of little hope for the software industry. He confronts the reader with the following statement, at some stage:

"We must learn to use mathematics in software development, but we need to question, and be prepared to discard, most of the methods that we have been discussing and promoting for all these years."

At the core of Parnas objections we find the contrast between the current ad-hoc (re)invention of mathematical concepts which are cumbersome and a burden to use and elegant (and therefore useful) concepts which are neglected, often for cultural or (lack of) background reasons.

The question is: what is it that tells "good and "bad" methods apart? As Parnas writes, *there is a disturbing gap between software development and traditional engineering disciplines*. In such disciplines one finds a successful, well-established mathematical background taught regularly at every higher-education institute, essentially made of calculus (derivatives and integrals), linear algebra and probability theory. This raises another question: can one hope to share such a successful tradition in the computing field, or is this definitely a different kind of science, hostage of formal logics and set theory?

For quite some time this has been the common understanding but, with the advent of quantitative formal methods, things are changing. For instance, there is a trend towards the explicit use of matrix operations and linear algebra techniques in computing, driven by disparate research interests. In the area of evaluating high-availability standby redundant clusters, for instance, Distefano *et al* [2] develop a technique resorting to Kronecker sums and products. In the field of quantum programming and semantics of probabilistic programs, Sernadas *et al* [3] use linear algebra techniques in considering a probabilistic program to be a linear transformation over a suitable vector space. In the field of data mining, the authors of this text show in [4] how to perform some OLAP operations solely based on linear algebra operations. Trčka [5] presents a unifying matrix approach to the notions of strong, weak, and branching bisimulation ranging from labeled transition systems to Markov reward chains. And many other examples could be given, which are omitted for space economy.

Why is linear algebra so appealing? In this extended abstract, we start by briefly investigating how often linearity (understood in the broad sense) is present in models of computation, and therefore underlying the notion of a software component. In fact, most computing models are linear in some sense. This leads into Kleene and regular algebras and their matrix extensions [6,7]. As a special case we will find the classical interpretation of non-deterministic computation captured by binary relations, a way of doing algebraic logic which is a century and a half old [8]. This extends to the very general concept of an allegory [9] which, in general, has to do with matrices whose data values form locales.

Finally, we will draw attention to the categorical, matricial machine based theory of concurrency developed by Bloom *et al* [10]. Work in this vein can be traced much earlier, back to Conway's work on *regular algebras* [6] and regular algebras of matrices, so elegantly presented in [7, Chap. 10].

This leads into our own work in developing a generic, typed approach to linear algebra [11], based on the *biproduct* concept [12] and aiming at capturing the best of both ends: type systems from the computing side and the powerful blocked matrix algebra from the mathematics side.

2 Linear Algebras of Machines

Any abstract notion of a computation is bound to encompass two kernel operations, one multiplicative $x \cdot y$ (usually abbreviated to xy) capturing **sequencing** and the other additive $(x + y)$ capturing **choice** (alternation). The additive unit 0 will mean **death** and the multiplicative unit 1 will mean **skip** (do nothing).

One is thus lead into a semiring $(S, +, \cdot, 0, 1)$ of computations, meaning that $(S, \cdot, 1)$ is a monoid, $(S, +, 0)$ is a Abelian (commutative) monoid, 0 annihilates $(\cdot)$, $x\ 0 = 0\ x = 0$, and $(\cdot)$ distributes over $(+)$:

$$x(y + z) = xy + xz \quad , \qquad (y + z)x = yx + zx \tag{1}$$

Property (1) is at the heart of building matrices M of computations over semiring S: it is necessary and sufficient for matrix multiplication, also denoted by $(\cdot)$, to be associative [9] $M \cdot (N \cdot R) = (M \cdot N) \cdot R$ and form a monoid too.

Such is the beginning of any linear algebra of computations. In the relational approach to non-determinism, for instance, S is the Boolean algebra of truth values and matrices are relations. The set-valued function approach also falls in this bin, since the powerset lattice of outputs is also a Boolean algebra. For *aficionados* of relational methods and inequational reasoning, both situations above lead to allegories, as shown in [9]. Also the case for the LTS model in [5].

A similar, and perhaps better standpoint for building up the same framework is to regard S as a regular algebra $(S, +, \cdot, \leq, 0, 1)$ [6] and scale this up to the regular algebra of S-valued matrices [7]. The advantage of this approach is that of bringing a number of Galois connections to surface, which prove very useful in the reasoning, as [7] amply shows. For instance, (1) does not need be postulated in this approach: it arises from $(x\cdot)$ and $(\cdot x)$ being lower adjoints.

Bloom *et al* [10] go very close to building a linear algebra of components by developing a matricial, machine-based theory of concurrency where S is the semiring $\mathcal{L}(X^*)$ of subsets of words on a finite alphabet X. They provide block-definitions for the iteration M^* of a square matrix M and the feedback $M^\uparrow$ of matrix M, on some conditions. They further define machine composition, machine target and source tupling, and machine Kronecker product (termed *shuffle product*). Via these constructions they obtain a compositional and modular approach to build complex machines, whose behavioural semantics are given via a functor associating each machine with a matrix on a category of $\mathcal{L}(X^*)$-valued matrices.

3 Typed Matrices for Blockwise Abstraction

Computer scientists tend to regard matrices as rectangular shaped data structures, bidimensional arrays, lists of lists, and the like. Mathematicians tend to regard them as linear transforms, i.e. vector-to-vector operations. Yet matrices are abstract entities independent of either such views: they can be regarded as arrows of particular categories, whereby they become *typed*. This answers questions such as: what is the type of a matrix? What are their basic *constructors*? In what measure are these related to standard matrix operations and algebra?

By studying categories of matrices [12] we have defined typed, algebraically rich constructors [11] repairing the lack mentioned. Backhouse [7] writes that matrices are a way of compacting sets of equations into single equations which *"is a tremendous improvement in concision that does not incur any loss of precision!"*. In [11] we show how the very general concept of a *biproduct* promotes individual values to blocks and value-level operations to block-level operations, after all the great conceptual advantage offered by matrix notation.

Below we draw a diagram in which arrows of shape $m \xrightarrow{M} n$ represent matrices with m (input) columns and n (output) rows. Given matrices R, S, U, V, other universal matrices are brought to light explaining how the construction of a blocked matrix (either by juxtaposition $[R|S]$ or stacking $\left[\frac{R}{S}\right]$) is made from the original R and S by composition with biproduct (elementary) matrices π_1 and π_2 and adding the results:

$$[R|S] = R \cdot \pi_1 + S \cdot \pi_2$$

$$\left[\frac{U}{V}\right] = i_1 \cdot U + i_2 \cdot V$$

$$\begin{tikzcd}
 & m & \\
n \arrow[ur, "R"] \arrow[r, shift right, "i_1"'] & n+p \arrow[u, "{[R|S]}"'] \arrow[l, shift right, "\pi_1"'] \arrow[r, shift left, "\pi_2"] & p \arrow[ul, "S"'] \arrow[l, shift left, "i2"] \\
 & t \arrow[ul, "U"] \arrow[ur, "V"'] \arrow[u, "{\left[\frac{U}{V}\right]}"] &
\end{tikzcd}$$

The composition operator $(\cdot)$ is the usual matrix multiplication, $(+)$ is matrix addition and the elementary matrices are $\pi_1 = \begin{bmatrix}1 & 0\end{bmatrix}$ and $\pi_2 = \begin{bmatrix}0 & 1\end{bmatrix}$, taking 1 and 0 as the identity and zero matrices of suitable dimensions. Matrices i_1 and i_2 are obtained by transposing π_1 and π_2, respectively. More elaborate biproducts can be defined catering for special matrix operations, for instance Gauss elimination

[11]. More details about the approach can be consulted in [11] where it is used to formally derive blocked matrix algorithms.

4 Postlude

We have shown that there is significant foundational work on what could lead to a linear algebra approach to software components. Should these be weighted automata [13] or machines in the sense of Bloom *et al* [10], then [11] offers a constructive, fully typed approach to such matrix models. However abstract or primitive these may look to practitioners, they promise the calculation style which is the hallmark of engineering disciplines. Back to Parnas reflections [1], engineers rarely use the phrase *proof of correctness*. Instead, they *calculate* properties of the products they build. Linear algebras are especially apt for calculation.

Let us dare going in this way and lay proper foundations for the emerging disciplines of software architecture and component-ware. Otherwise we shall be wasting our time.

References

1. Parnas, D.L.: Really rethinking 'formal methods'. IEEE Computer 43(1), 28–34 (2010)
2. Distefano, S., Longo, F., Scarpa, M.: Availability assessment of ha standby redundant clusters. In: 29th IEEE Int. Symp. on Reliable Distributed Systems (2010)
3. Sernadas, A., Ramos, J., Mateus, P.: Linear algebra techniques for deciding the correctness of probabilistic programs with bounded resources. Technical report, TU Lisbon, Short paper, LPAR, Doha, Qatar (November 22-27, 2008)
4. Macedo, H., Oliveira, J.: Can we teach computers to generate fast OLAP code? Technical note (May 2010), `http://wiki.di.uminho.pt`
5. Trcka, N.: Strong, weak and branching bisimulation for transition systems and Markov reward chains: A unifying matrix approach. In: Andova, S.E. (ed.) Proc. 1st Workshop on Quantitative Formal Methods: Theory and Applications (December 2009)
6. Conway, J.: Regular Algebra and Finite Machines. Chap.& Hall, London (1971)
7. Backhouse, R.: Mathematics of Program Construction, 608 pages. Univ. of Nottingham (2004) Draft of book in preparation
8. Maddux, R.: The origin of relation algebras in the development and axiomatization of the calculus of relations. Studia Logica 50, 421–455 (1991)
9. Freyd, P., Scedrov, A.: Categories, Allegories. Mathematical Library, vol. 39. North-Holland, Amsterdam (1990)
10. Bloom, S.L., Sabadini, N., Walters, R.F.C.: Matrices, machines and behaviors. Applied Categorical Structures 4, 343–360 (1996), doi:10.1007/BF00122683
11. Macedo, H., Oliveira, J.: Matrices As Arrows! A Biproduct Approach to Typed Linear Algebra. In: Bolduc, C., Desharnais, J., Ktari, B. (eds.) MPC 2010. LNCS, vol. 6120, pp. 271–287. Springer, Heidelberg (2010)
12. MacLane, S.: Categories for the Working Mathematician (Graduate Texts in Mathematics). Springer, Heidelberg (September 1998)
13. Bonchi, F., Silva, A., Bonsangue, M., Rutten, J.: Quantitative Kleene coalgebras. In: Information and Computation. Academic Press, London (November 2010) ISSN: 0890-5401

Author Index