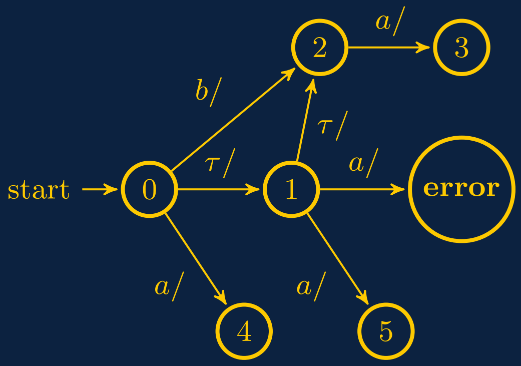
LNCS 8050

Tutorial

Zhiming Liu
Jim Woodcock
Huibiao Zhu (Eds.)

Unifying Theories of Programming and Formal Engineering Methods

International Training School on Software Engineering
Held at ICTAC 2013
Shanghai, China, August 2013, Advanced Lectures



Springer

Zhiming Liu   Jim Woodcock   Huibiao Zhu (Eds.)

Unifying Theories of Programming and Formal Engineering Methods

International Training School on Software Engineering
Held at ICTAC 2013
Shanghai, China, August 26-30, 2013
Advanced Lectures

Springer

Volume Editors

Zhiming Liu
United Nations University
International Institute for Software Technology
P.O. Box 3058, Macau, China
E-mail: z.liu@iist.unu.edu

Jim Woodcock
University of York
Department of Computer Science
Deramore Lane, York YO10 5GH, UK
E-mail: jim@cs.york.ac.uk

Huibiao Zhu
East China Normal University
Software Engineering Institute
3663 Zhongshan Road (North), Shanghai 200062, China
E-mail: hbzhu@sei.ecnu.edu.cn

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-39720-2 e-ISBN 978-3-642-39721-9
DOI 10.1007/978-3-642-39721-9
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2013943600

CR Subject Classification (1998): F.3, D.2.4, F.4, I.1, I.2.2-3, F.1, D.2, I.6

LNCS Sublibrary: SL 1 – Theoretical Computer Science and General Issues

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

Printed on acid-free paper

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

Preface

This volume contains the lecture notes of the courses given at the ICTAC 2013 Software Engineering School on Unifying Theories of Programming and Formal Engineering Methods, held during August 26–30, 2013, in Shanghai. East China Normal University, UNU-IIST, and the University of York organized the school as part of the celebrations of the 70th birthday of He Jifeng. There were two associated events:

- *Essays in Honor of He Jifeng on the Occasion of his 70th Birthday.* Papers presented at a Symposium held in Shanghai during September 1–3, 2013. LNCS volume 8051, Springer 2013.
- Proceedings of the International Colloquium on Theoretical Aspects of Computing. Held in Shanghai during September 4–6, 2013.

The school is aimed at postgraduate students, researchers, academics, and industrial engineers who are interested in the state of the art in *unifying theories of programming and formal engineering methods.* This volume contains the lecture notes of the five courses. The common themes of the courses include the design and use of formal models and specification languages with tool support. System wide, the courses cover component-based and service-oriented systems, real-time systems, hybrid systems, and cyber physical systems. Techniques include inductive theorem proving, model checking, correct by construction through refinement and model transformations, synthesis and computer algebra. Two of the courses are explicitly related to Hoare and He's unifying theories. No previous knowledge of the topics involved is assumed.

We would like to acknowledge sponsorship from the following organizations:

- East China Normal University
- United Nations University - International Institute for Software Technology
- University of York

Lecturers and Editors

ETHAN K. JACKSON is Researcher in The Research in Software Engineering (RiSE) Group at Microsoft Research. His work focuses on next-generation formal specification languages for model-based development with an emphasis on automated synthesis. He is the developer of the FORMULA language, which has been applied to software, cyber-physical, and biological systems. Ethan received his PhD in Computer Science from Vanderbilt University in 2007 and his BS in Computer Engineering from the University of Pittsburgh in 2004. He joined Microsoft Research in 2007.

KIM G. LARSEN is Professor in the Department of Computer Science at Aalborg University, director of CISS (Center for Embedded Software Systems), as well as director of the Innovation Network InfinIT. He is also co-director of the VKR Center of Excellence MT-LAB and director of the new Danish-Chinese Basic Research Center IDEA4CPS. Currently, he is investing substantial effort in a number of European projects devoted to model-based development: MBAT, CRAFTERS, RECOMP, SENSATION and CASSTINGS. Kim G. Larsen's research includes modeling, verification, performance analysis of real-time and embedded systems with applications to concurrency theory and model checking. In particular he is prime investigator of the real-time verification UPPAAL as well as its various new branches of the tool targeted toward optimization, testing, synthesis, and compositional analysis.

ZHIMING LIU is Senior Research Fellow at the United Nations University - International Institute for Software Technology (UNU-IIST). He is the Head of the Program on Information Engineering and Technology in Healthcare. His is known for his work on the transformational approach for real-time and fault-tolerant system specification and verification, and the rCOS Formal Model-Driven Software Engineering Method. He is currently leading a research group of a dozen young researchers working in the areas of formal model-driven software engineering methods, program static analysis, and applications in electronic health record-based healthcare applications.

JIM WOODCOCK is Head of the Department of Computer Science at the University of York, where he is also Professor of Software Engineering. His research interests in software engineering include methods and tools for specification, refinement, and proofs of correctness. He is currently an investigator in the European COMPASS project on comprehensive modeling of advanced systems of systems. The COMPASS Modeling Language includes a combination of rich state, concurrency, communication, time, and object orientation. The formal semantics is given in Unifying Theories of Programming, where each individual paradigm is dealt with as a separate theory and linked into a unified language design. Jim Woodcock is a Fellow of the British Computer Society and a Fellow of the Royal Academy of Engineering.

NAIJUN ZHAN is Full Professor at the Institute of Software, Chinese Academy of Sciences, where he is also the Deputy Director of State Key Laboratory of Computer Science. His research interests in formal methods and software engineering include formal techniques for the design of real-time and hybrid systems, program verification, modal and temporal logics, process algebra, theoretical foundations of component and object systems.

HUIBIAO ZHU is Professor of Computer Science at Software Engineering Institute, East China Normal University, also Executive Deputy Director of Shanghai Key Laboratory of Trustworthy Computing. He earned his PhD in Formal Methods from London South Bank University in 2005. He has studied various semantics and their linking theories for Verilog, SystemC, Web services and probability system. Currently, he is the Chinese PI of the Sino-Danish Basic Research Center IDEA4CPS.

Lecture Courses

Course 1: FORMULA 2.0: A Language for Formal Specifications. Ethan Jackson gives this course. It is on the specification language FORMULA 2.0. This is a novel formal specification language based on open-world logic programs and behavioral types. Its goals are (1) succinct specifications of domain-specific abstractions and compilers, (2) efficient reasoning and compilation of input programs, (3) diverse synthesis and fast verification. A unique approach is taken toward achieving these goals: Specifications are written as strongly typed open-world logic programs. They are highly declarative and easily express rich synthesis/verification problems. Automated reasoning is enabled by efficient symbolic execution of logic programs into constraints. This tutorial introduces the FORMULA 2.0 language and concepts through a series of small examples.

Course 2: Model-Based Verification, Optimization, Synthesis and Performance Evaluation of Real-Time Systems. Kim Larsen teaches this series of lectures. It aims at providing a concise and precise traveller's guide, phrase book or reference manual to the timed automata modeling formalism introduced by Alur and Dill. The course gives comprehensive definitions of timed automata, priced (or weighted) timed automata, timed games, stochastic timed automata and highlights a number of results on associated decision problems related to model checking, equivalence checking, optimal scheduling, the existence of winning strategies, and then statistical model checking.

Course 3: rCOS: Defining Meanings of Component-Based Software Architectures. In this course, Zhiming Liu teaches the rCOS method for component-based software development. Model-driven software development is nowadays seen as a mainstream methodology. In the software engineering community, it is a synonym of the OMG model-driven architecture (MDA). However, in the formal method community, model-driven development is broadly seen as model-based techniques for software design and verification. The method aims to bridge the gap between formal techniques, together with their tools, and their potential support to practical software development. To this end the course introduces the rCOS definition of the meanings of component-based software architectures, and shows how software architectures are formally modeled, designed, and verified in a model-driven engineering development process.

Course 4: Unifying Theories of Programming in Isabelle. This course is given by Jim Woodcock and Simon Foster and it introduces the two most basic theories in Hoare and He's Unifying Theories of Programming and their mechanization in the Isabelle interactive theorem prover. The two basic theories are the relational calculus and the logic of designs (pre-postcondition pairs). The course introduces a basic nondeterministic programming language and the laws of programming in this language based on the theory of designs. The other part of the course is about theory mechanization in Isabelle/HOL, and shows how the theorem prover is used to interpret the theory of designs of UTP.

Course 5. Formal Modeling, Analysis and Verification of Hybrid Systems. This course is given by Naijun Zhan. It introduces a systematic approach to formal modeling, analysis and verification of hybrid systems. Hybrid system is modeled using Hybird CSP (HCSP), an extension of Hoare's CSP. Then for specification and verification, Hoare logic is extended to Hybrid Hoare Logic (HHL). For deductive verification of hybrid systems, a complete approach is used to generate polynomial invariants for polynomial hybrid systems. The course also presents an implementation of a theorem prover for HHL. Real-time application case studies are used to demonstrate the language, the verification techniques, and tool support. The Chinese High-Speed Train Control System at Level 3 (CTCS-3) in particular is a real application. Furthermore, an example is given to show how, based on the invariant generation technique and using constraint solving, to synthesize a switching logic for a hybrid system to meet a given safety and liveness requirement.

June 2013

Zhiming Liu
Jim Woodcock
Huibiao Zhu

Organization

Coordinating Committee

Zhiming Liu	UNU-IIST, Macau, SAR China
Jim Woodcock	University of York, UK
Min Zhang	East China Normal University, China
Huibiao Zhu	East China Normal University, China

Local Organization

Mingsong Chen, Jian Guo, Xiao Liu, Geguang Pu, Fu Song, Min Zhang
East China Normal University

Table of Contents

rCOS: Defining Meanings of Component-Based Software Architectures

Ruzhen Dong[1,2], Johannes Faber[1], Wei Ke[3], and Zhiming Liu[1]

[1] United Nations University – International Institute for Software Technology, Macau
{ruzhen,jfaber,z.liu}@iist.unu.edu
[2] Dipartmento di Informatica, Università di Pisa, Italy
[3] Macao Polytechnic Institute, Macau
wke@ipm.edu.mo

Abstract. Model-Driven Software Development is nowadays taken as a mainstream methodology. In the software engineering community, it is a synonym of the OMG *Model-Driven Architecture* (MDA). However, in the formal method community, model-driven development is broadly seen as model-based techniques for software design and verification. Because of the difference between the nature of research and practical model-driven software engineering, there is a gap between formal techniques, together with their tools, and their potential support to practical software development. In order to bridge this gap, we define the meanings of component-based software architectures in this chapter, and show how software architectures are formally modeled in the formal model-driven engineering method rCOS. With the semantics of software architecture components, their compositions and refinements, we demonstrate how appropriate formal techniques and their tools can be applied in an MDA development process.

Keywords: Component-Based Architecture, Object-Oriented Design, Model, Model Refinement, Model Transformation, Verification.

1 Introduction

Software engineering was born and has been growing up with the "software crisis". The root of the crisis is the inherent complexity of software development, and the major cause of the complexity "is that the machines have become several orders of magnitude more powerful" [18] within decades. Furthermore, ICT systems with machines and smart devices that are communicating through heterogeneous Internet and communication networks, considering integrated healthcare information systems and environment monitoring and control systems, are becoming more complex beyond the imagination of the computer scientists and software engineers in the 1980's.

1.1 Software Complexity

Software complexity was characterized before the middle of the 1990s in terms of four fundamental attributes of software [5–7]:

1. the *complexity of the domain application*,
2. the *difficulty of managing the development process*,
3. the *flexibility possible* through software,
4. and the *problem of characterizing the behavior* of software systems [5].

This characterization remains sound, but the extensions of the four attributes are becoming much wider.

The first attribute focuses on the difficulty of understanding the application domain (by the software designer in particular), capturing and handling the ever-changing requirements. It is even more challenging when networked systems support collaborative workflows involving many different kinds of stakeholders and end users across different domains. Typical cases are in healthcare applications, such as telemedicine, where chronic conditions of patients on homecare plans are monitored and tracked by different healthcare providers. In these systems, requirements for safety, privacy assurances and security are profound too.

The second attribute concerns the difficulty to define and manage a development process that has to deal with changing requirements for a software project involving a large team comprising of software engineers and domain experts, possibly in different geographical places. There is a need of a defined development process with tools that support collaboration of the team in working on shared software artifacts.

The third is about the problem of making the right design decisions among a wide range of possibilities that have conflicting features. This includes the design or reuse of the software architecture, algorithms and communication networks and protocols. The design decisions have to deal with changing requirements and aiming to achieve the optimal performance to best support the requirements of different users.

The final attribute of software complexity pinpoints the difficulty in understanding and modeling the semantic behavior of software, for analysis, validation and verification for correctness, as well as reliability assurance. The semantic behavior of modern *software-intensive systems* [63], which we see in our everyday life, such as in transportation, health, banking and enterprise applications, has a great scale of complexity. These systems provide their users with a large variety of *services* and *features*. They are becoming increasingly *distributed*, *dynamic* and *mobile*. Their components are *deployed* over large networks of *heterogeneous platforms*. In addition to the complexity of functional structures and behaviors, modern software systems have complex aspects concerning *organizational structures* (i.e., *system topology*), *adaptability*, *interactions*, *security*, *real-time* and *fault-tolerance*. Thus, the availability of models for system architecture components, their interfaces, and compositions is crucially important.

Complex systems are open to total breakdowns [53], and consequences of system breakdowns are sometimes catastrophic and very costly, e.g., the famous Therac-25 Accident 1985-1987 [41], the Ariane-5 Explosion in 1996 [56], and the Wenzhou High Speed Train Collision.[1] Also the software complexity attributes are the main source of *unpredictability* of software projects, software projects fail

[1] `http://en.wikipedia.org/wiki/Wenzhou_train_collision`

due to our failure to master the complexity [33]. Given that the global economy as well as our everyday life depends on software systems, we cannot give up advancing the theories and the engineering methods to master the increasing complexity of software development.

1.2 Model-Driven Development

The *model-driven architecture* (MDA) [52, 60, 63] approach proposes building of system models in all stages of the system development as the key engineering principle for mastering software complexity and improving dependability and predictability. The notion of software architectures is emphasized in this approach, but it has not been precisely defined. In industrial project development, the architecture of a system at a level of abstraction is often represented by diagrams with "boxes" and "links" to show parts of the systems and their linkages, and sometimes these boxes are organized into a number of "layers" for a "layered architecture". There even used to be little informal semantic meaning for the boxes and links. This situation has improved since the introduction of the Unified Modeling Language (UML) in which boxes are defined as "objects" or "components", and links are defined as "associations" or "interfaces". These architectural models are defined as both *platform independent models* (PIM) and *platform specific models* (PSM), and the mapping from a PIM to a PSM is characterized as a *deployment model.*

MDA promotes in software engineering the principles of *divide and conquer* by which an architectural component is hierarchical and can be divided into subcomponents; *separation of concerns* that allows a component to be described by models of different viewpoints, such as static component and class views, interaction views and dynamic behavioral views; and *information hiding by abstractions* so that software models at different stages of development only focus on the details relevant to the problem being solved at that stage.

All the different architectural models and models of viewpoints are important when defining and managing a development process [43]. However, the semantics of these models is largely left to the user to understand, and the integration and transformation of these models are mostly syntax-based. Hence, the tools developed to support the integration and transformation cannot be integrated with tools for verification of semantic correctness and correctness preserving transformations [46].

For MDA to support a seamless development process of model decomposition, integration, and refinement, there is a need of formal semantic relations between models of different viewpoints of the architecture at a certain level, and the refinement/abstraction relation between models at different levels of abstraction. It is often the case that a software project in MDA only focuses on the grand levels of abstraction — *requirements*, *design*, *implementation* and *deployment*, without effectively supporting refinement of the requirements and design models, except for some model transformations based on design patterns. This is actually the reason why MDA has failed to demonstrate the potential advantages of *separation of concerns*, *divide and conquer* and *incremental development* that it promises. This lack of semantic relations between models as well as the lack of techniques and tools for semantics-preserving model transformations is also an essential barrier for MDA to realize its full potential in improving safety and predictability of software systems.

1.3 Formal Methods in Software Development

Ensuring semantic correctness of computer systems is the main purpose of using formal methods. A *formal method* consists of a body of techniques and tools for the *specification*, *development*, and *verification* of programs of a certain paradigm, such as sequential or object-oriented procedural programs, concurrent and distributed programs and now web-services. Here, a specification can be a description of an abstract model of the program or the specification of desirable properties of the program in a formally defined notation. In the former case, the specification notation is also often called a modeling language, though a modeling language usually includes graphic features. Well-known modeling/specification languages include CSP [28], CCS [50], the Z-Notation [58], the B-Method [1,2], VDM [34], UNITY [9], and TLA+ [38]. In the latter case, i.e., the specification of program properties, these desirable properties are defined on a computational model of the executions of the system, such as state machines or transition systems. Well-known models of this kind include the *labeled transition systems* and the *linear temporal logic* (LTL) of Manna and Pnueli [49], which are also used in verification tools like Spin [31] and, in an extended form, Uppaal.[2]

The techniques and tools of a formal method are developed based on a mathematical theory of the execution or the behavior of programs. Therefore, we define a *formal method* to include a *semantic theory* as well as the *techniques* and *tool support* underpinned by the theory for *modeling*, *design*, *analysis*, and *verification* of programs of a defined programming paradigm. It is important to note that the semantic theory of a formal method is developed based on the fundamental theories of *denotational semantics* [59], *operational semantics* [54], and *axiomatic semantics* (including algebraic semantics) [17,27] of programming. As they are all used to define and reason about behavior of programs, they are closely related [51], and indeed, they can be "unified" [29].

In the past half a century or so, a rich body of formal theories and techniques have been developed. They have made significant contribution to program behavior characterization and understanding, and recently there has been a growing effort in development of tool support for verification and reasoning. However, these techniques and tools, of which each has its community of researchers, have been mostly focusing on models of individual viewpoints. For examples, type systems are used for data structures, Hoare Logic for local functionality, process calculi (e.g., CSP and CSS) and I/O automata [48] for interaction and synchronization protocols. While process calculi and I/O automata are similar from the perspective of describing the interaction behavior of concurrent and distributed components, the former is based on the observation of the global behavior of

[2] `http://www.uppaal.org`

interaction sequences, and the latter on the observation of local state transitions caused by interaction events. Processes calculi emphasize on support of algebraic reasoning, and automata are primarily used for algorithmic verification techniques, i.e., model checking [15, 55].

All realistic software projects have design concerns on all viewpoints of data structures, local sequential functionalities, and interactions. The experience, e.g., in [32], and investigation reports on software failures, such as those of the Therac-25 Accident in 1985–1987 [41] and the Ariane-5 Explosion in 1996 [56], show that the cause of a simple bug that can lead to catastrophic consequences and that *ad hoc* application of formal specification and verification to programs or to models of programs will not be enough or feasible to detect and remove these causes. Different formal techniques that deal with different concerns more effectively have to be systematically and consistently used in all stages of a development process, along with safety analysis that identifies risks, vulnerabilities, and consequences of possible risk incidents. There are applications that have extra concerns of design and verification, such as real-time and security constraints. Studies show that models with these extra functionalities can be mostly treated by model transformations into models for requirements without these concerns [44].

1.4 The Aim and Theme of rCOS

The aim of the rCOS method is to bridge the gap sketched in the previous sections by defining the unified meanings of component-based architectures at different levels of abstraction in order to support seamless integration of formal methods in modeling software development processes. It thus provides support to MDA with formal techniques and tools for predictable development of reliable software. Its scope covers theories, techniques, and tools for modeling, analysis, design, verification and validation. A distinguishing feature of rCOS is the formal model of system architecture that is essential for model compositions, transformations, and integrations in a development process. This is particularly the case when dealing with safety critical systems (and so must be shown to satisfy certain properties before being commissioned), but beyond that, we promote with rCOS the idea that formal methods are not only or even mainly for producing software that is safety critical: they are just as much needed when producing a software system that is too complex to be produced without tool assistance. As it will be shown in this chapter, rCOS systematically addresses these complexity problems by dealing with architecture at a large granularity, compositionality, and separation of concerns.

1.5 Organization

Following this introduction section, we lay down the semantic foundation in Sect. 2 by developing a general model of *labeled transition systems* that combines the local computation (including structures and objects) in a transition and the dynamic behavior of interactions. We propose a *failure-divergence semantics* and a *failure-divergence refinement* relation between transition systems following the techniques of CSP [57]. Then in Sect. 3-5, we introduce the specification of *primitive closed components*, *primitive open components* and *processes* that are the basic architectural components in rCOS. Each of the different kinds of components is defined by their corresponding label transition systems. Models at different levels of abstraction, including *contracts* and *publications* for different purposes in a model-driven development process, are defined and their relations are studied. Section 6 defines the architectural operators for composing and adapting components. These operations on component specifications show how internal autonomous transitions are introduced and how they cause nondeterminism that we characterized in the general labeled transition systems. These operators extend and generalize the limited plugging, disjoint union and gluing operators we defined in the original components. The model also unifies the semantics and compositions of components and processes. A *general* component can exhibit both passive behavior of receiving service requests and actively invoke services from the environment. This is a major extension, but it preserves the results that we have developed for the original rCOS model. However, this extension still needs more detailed investigation in the future, including their algebraic properties. Section 7 is about a piece of work on an interface model of rCOS components. The aim is to propose an input-deterministic model of component interfaces for better composability checking, and to give a more direct description of provided and required protocols of components. There, we define a partial order, called *alternative refinement*, among component interface models. The results are still preliminary, and interesting further research topics are thus pointed out. Concluding remarks are given and future work is discussed in Sect. 8.

2 Unified Semantics of Sequential Programming

The rCOS method supports programming software components that exhibit interacting behavior with the environment as well as local data functionality through the executions of operations triggered by interactions. The method supports *interoperable compositions* of components for that the local data functionality are implemented in different programming paradigms, including modular, procedural and object-oriented programming. This requires a unified semantic theory of models of programs. To this end, rCOS provides a theory of relational semantics for object-oriented programming, in which the semantic theories of modular and procedural programming are embedded as sub-theories. This section first introduces a theory of sequential programs, which is then extended by concepts for object-oriented and reactive systems.

To support model-driven development, models of components built at different development stages are related so that properties established for a model at a higher level of abstraction are preserved by its lower level *refined models*. Refinement of components is also built on a refinement calculus of object-oriented programs.

2.1 Designs of Sequential Programs

We first introduce a unified theory of imperative sequential programming. In this programming paradigm, a program P is defined by a set of *program variables*, called the *alphabet* of P, denoted by αP, and a program command c written in the following syntax, given as a BNF grammar,

$$c ::= x := e \mid c; c \mid c \lhd b \rhd c \mid c \sqcap c \mid b * c \qquad (1)$$

where e is an expression and b a boolean expression; $c_1 \lhd b \rhd c_2$ is the conditional choice equivalent to "if b then c_1 else c_2" in other programming languages; $c \sqcap c$ is the *non-deterministic choice* that is used as an abstraction mechanism; $b * c$ is iteration equivalent to "while b do c".

A sequential program P is regarded as a closed program such that for given initial values of its variables (that form an *initial state*), the execution of its command c will change them into some possible final values, called the *final state* of the program, if the execution terminates. We follow UTP [29] to define the semantics of programs in the above simple syntax as relations between the initial and final states.

States. We assume an infinite set of names $\mathcal{X}$ representing *state variables* with an associated value space $\mathcal{V}$. We define a *state* of $\mathcal{X}$ as a function $s : \mathcal{X} \rightarrow \mathcal{V}$ and use Σ to denote the set of all states of $\mathcal{X}$. This allows us to study all the programs written in our language. For a subset X of $\mathcal{X}$, we call Σ_X the restrictions of Σ on X *the states of* X; an element of this set is called *state over* X. Note that state variables include both variables used in programs and auxiliary variables needed for defining semantics and specifying properties of programs. In particular, for a program, we call $\Sigma_{\alpha P}$ the *states of program* P.

For two sets X and Y of variables, a state s_1 over X and a state s_2 over Y, we define $s_1 \oplus s_2$ as the state s for which $s(x) = s_1(x)$ for $x \in X$ but $x \notin Y$ and $s(y) = s_2(y)$ for $y \in Y$. Thus, s_2 overwrites s_1 in $s_1 \oplus s_2$.

State Properties and State Relations. A state property is a subset of the states Σ and can be specified by a predicate over $\mathcal{X}$, called a *state predicate.* For example, $x > y + 1$ defines the set of states s for that $s(x) > s(y) + 1$ holds. We say that a state s satisfies a predicate F, denoted by $s \models F$, if it is in the set defined by F.

A *state relation* R is a relation over the states Σ, i.e., a subset of the Cartesian product $\Sigma \times \Sigma$, and can be specified by a predicate over the state variables $\mathcal{X}$ and their primed version $\mathcal{X}' = \{x' \mid x \in \mathcal{X}\}$, where $\mathcal{X}'$ and $\mathcal{X}$ are disjoint sets of names. We say that a pair of states (s, s') satisfies a relation predicate $R(x_1, \ldots, x_k, y'_1, \ldots, y'_n)$ if

$$R(s(x_1)/x_1, \ldots, s(x_k)/x_k, s'(y_1)/y'_1, \ldots, s'(y_n)/y'_n)$$

holds, denoted by $(s, s') \models R$. Therefore, a relational predicate specifies a set of possible state changes. For example, $x' = x+1$ specifies the possible state changes from any initial state to a final state in which the value of x is the value of x in the initial state plus 1. However, $x' \geq x + 1$ defines the possible changes from an initial state to a state in which x has a value not less than the initial value plus 1. A state predicate and a relational predicate only constrain the values of variables that occur in the predicates, leaving the other variables to take values freely. Thus, a state predicate F can also be interpreted as a relational predicate such that F holds for (s, s') if s satisfies F. In addition to the conventional propositional connectors $\vee$, $\wedge$ and $\neg$, we also define the sequential composition of relational predicates as the composition of relations

$$R_1; R_2 \mathrel{\widehat{=}} \exists x_0 \bullet R_1(x_0/x') \wedge R_2(x_0/x), \tag{2}$$

where x_0 is a vector of state variables; x and x' represent the vectors of all state variables and their primed versions in R_1 and R_2; and the substitutions are element-wise substitutions. Therefore, a pair of states (s, s') satisfies $R_1; R_2$ iff there exists a state s_0 such that (s, s_0) satisfies R_1 and (s_0, s') satisfies R_2.

Designs. A semantic model of programs is defined based on the way we observe the execution of programs. For a sequential program, we observe which possible final states a program execution reaches from an initial state, i.e., the relation between the starting states and the final states of the program execution.

Definition 1 (Design). *Given a finite set α of program variables (as the alphabet of a program in our interest), a state predicate p and a relational predicate R over α, we use the pair $(\alpha, p \vdash R)$ to represent a program* **design**. *The relational predicate $p \vdash R$ is defined by $p \Rightarrow R$ that specifies a program that starts from an initial state s satisfying p and if its execution terminates, it terminates in a state s' such that $(s, s') \models R$.*

Such a design does not observe the *termination* of program executions and it is a model for reasoning about *partial correctness*. When the alphabet is known, we simply denote the design by $p \vdash R$. We call p the *precondition* and R the *postcondition* of the design.

To define the semantics of programs written in Syntax (1), we define the operations on designs over the same alphabet. In the following inductive definition, we use a simplified notation to assign design operations to program constructs. Note that on the left side of the definition, we mean the program symbols while the right side uses relational operations over the corresponding designs of a program, i.e., we identify programs with a corresponding design.

$$\begin{aligned} x := e &\mathrel{\widehat{=}} \mathit{true} \vdash x' = e \wedge \bigwedge\nolimits_{y \in \alpha, y \not\equiv x} y' = y, \\ c_1; c_2 &\mathrel{\widehat{=}} c_1; c_2 \\ c_1 \lhd b \rhd c_2 &\mathrel{\widehat{=}} b \wedge c_1 \vee \neg b \wedge c_2, \\ c_1 \sqcap c_2 &\mathrel{\widehat{=}} c_1 \vee c_2, \\ b * c &\mathrel{\widehat{=}} (c; b * c) \lhd b \rhd \mathbf{skip}, \end{aligned} \tag{3}$$

where we have **skip** $\widehat{=}$ *true* $\vdash \bigwedge_{x\in\alpha}(x' = x)$. We also define **chaos** $\widehat{=}$ *false* $\vdash$ *true*. In the rest of the paper, we also use *farmed designs* of the form $X : p \vdash R$ to denote that only variables in X can be changed by the design $p \vdash R$. So $x := e = \{x\} : \mathit{true} \vdash x' = e$.

However, for the semantics definition to be sound, we need to show that the set $\mathcal{D}$ of designs is closed under the operations defined in (3), i.e., the predicates on the right-hand-side of the equations are equivalent to designs of the form $p \vdash R$. Notice that the iterative command is inductively defined. Closure requires the establishment of a partial order $\sqsubseteq$ that forms a *complete partial order* (CPO) for the set of designs $\mathcal{D}$.

Definition 2 (Refinement of designs). *A design $D_l = (\alpha, p_l \vdash R_l)$ is a* refinement *of a design $D_h = (\alpha, p_h \vdash R_h)$, if*

$$\forall x, x' \bullet (p_l \Rightarrow R_l) \Rightarrow (p_h \Rightarrow R_h)$$

is valid, where x and x' represent all the state variables and their primed versions in D_l and D_h.

We denote the refinement relation by $D_h \sqsubseteq D_l$. The refinement relation says that any property satisfied by the "higher level" design D_h is preserved (or satisfied) by the "lower level" design D_l. The refinement relation can be proved using the following theorem.

Theorem 1. *$D_h \sqsubseteq D_l$ when*

1. *the pre-condition of the lower level is weaker: $p_h \Rightarrow p_l$, and*
2. *the post-condition of the lower level is stronger: $p_l \wedge R_l \Rightarrow R_h$.*

The following theorem shows that $\sqsubseteq$ is indeed a "refinement relation between programs" and forms a CPO.

Theorem 2. *The set $\mathcal{D}$ of designs and the refinement relation $\sqsubseteq$ satisfy the following properties:*

1. *$\mathcal{D}$ is closed under the sequential composition "$;$", conditional choice "$\lhd\, b \,\rhd$" and non-deterministic choice "$\sqcap$" defined in (3),*
2. *$\sqsubseteq$ is a partial order on the domain of designs $\mathcal{D}$,*
3. *$\sqsubseteq$ is preserved by sequential composition, conditional choice and non-deterministic choice, i.e., if $D_h \sqsubseteq D_l$ then for any D*

$$\begin{array}{ll} D; D_h \sqsubseteq D; D_l, & D_h; D \sqsubseteq D_l; D, \\ D_h \lhd b \rhd D \sqsubseteq D_l \lhd b \rhd D, & D_h \sqcap D \sqsubseteq D_l \sqcap D, \end{array}$$

4. *$(\mathcal{D}, \sqsubseteq)$ forms a CPO and the recursive equation $X = (D; X) \lhd b \rhd$ **skip** has a smallest fixed-point, denoted by $b * D$, which may be calculated from the bottom element* **chaos** *in $(\mathcal{D}, \sqsubseteq)$.*

For the proof of the theorems, we refer to the book on UTP [29]. D_1 and D_2 are *equivalent*, denoted as $D_1 = D_2$ if they refine each other, e.g., $D_1 \sqcap D_2 = D_2 \sqcap D_1$, $D_1 \lhd b \rhd D_2 = D_2 \lhd \neg b \rhd D_1$, and $D_1 \sqcap D_2 = D_1$ iff $D_1 \sqsubseteq D_2$. Therefore, the relation $\sqsubseteq$ is fundamental for the development of the refinement calculus to support correct by design in program development, as well as for defining the semantics of programs.

When refining a higher level design to a lower level design, more program variables are introduced, or types of program variables are changed, e.g., a set variable implemented by a list. We may also compare designs given by different programmers. Thus, we must relate programs with different alphabets.

Definition 3 (Data refinement). *Let $D_h = (\alpha_h, p_h \vdash R_h)$ and $D_l = (\alpha_l, p_l \vdash R_l)$ be two designs. $D_h \sqsubseteq D_l$ if there is a design $(\alpha_h \cup \alpha_l, \rho(\alpha_l, \alpha_h'))$ such that $\rho; D_h \sqsubseteq D_l; \rho$. We call ρ a* data refinement mapping.

Designs of Total Correctness. The designs defined above do not support reasoning about termination of program execution. To observe execution initiation and termination, we introduce a boolean state variable *ok* and its primed counterpart ok', and lift a design $p \vdash R$ to $\mathcal{L}(p \vdash R)$ defined below:

$$\mathcal{L}(p \vdash R) \mathrel{\widehat{=}} ok \wedge p \Rightarrow ok' \wedge R.$$

This predicate describes the execution of a program in the following way: if the execution starts successfully ($ok = true$) in a state s such that precondition p holds, the execution will terminate ($ok' = true$) in a state s' for which $R(s, s')$ holds. A design D is called a *complete correctness design* if $\mathcal{L}(D) = D$. Notice that $\mathcal{L}$ is a *healthy lifting function* from the domain $\mathcal{D}$ of partially correct designs to the domain of complete correct designs $\mathcal{L}(\mathcal{D})$ in that $\mathcal{L}(\mathcal{L}(D)) = \mathcal{L}(D)$. The refinement relation can be lifted to the domain $\mathcal{L}(\mathcal{D})$, and Theorem 1 and 2 both hold. For details of UTP, we refer to the book [29]. In the rest of the paper, we assume the complete correctness semantic model, and omit the lifting function $\mathcal{L}$ in the discussion.

Linking Theories. We can unify the theories of Hoare-logic [27] and the predicate transformer semantics of Dijkstra [17]. The Hoare-triple $\{p\}D\{r\}$ of a program D, which can be represented as a design according to the semantics given above, is defined to be $p \wedge D \Rightarrow r'$, where p and r are state predicates and r' is obtained from r by replacing all the state variables in r with their primed versions.

Given a state predicate r, the *weakest precondition* of the postcondition r for a design D, $wp(p \vdash R, r)$, is defined to be $p \wedge \neg(R; \neg r)$. Notice that this is a state predicate.

This unification allows the use of the laws in both theories to reason about program designs within UTP.

2.2 Designs of Object-Oriented Programs

We emphasize the importance of a semantic theory for concept clarification, development of techniques and tool support for *correct by design* and verification. The semantic theory presented in the previous section needs to be extended to define the concepts of classes, objects, methods, and OO program execution. The execution of an OO program is more complex than that of a traditional sequential program because the execution states have complex structures and properties. The semantics of OO programs has to cohesively define and treat

- the concepts of object heaps, stacks and stores,
- the problems of *aliasing*,
- subtyping and polymorphism introduced through the class inheritance mechanism, and
- dynamic typing of expression evaluation and dynamic binding of method invocation.

Without an appropriate definition of the execution state, the classic Hoare-logic cannot be used to specify OO program executions. Consider two classes C_1 and C_2 such that C_1 is a subclass of C_2 (denoted by $C_1 \preceq C_2$), and variables $C_1\ x_1$ and $C_2\ x_2$ of the classes, respectively. Assume a is an attribute of C_2 and thus also an attribute of C_1, the following Hoare-triple (confer previous section for representing Hoare-triples as designs) holds when x_1 and x_2 do not refer to the same object, i.e., they are not aliases of the same object, but does not necessarily hold if they refer to the same object:

$$\{x_2.a = 4\}\ x_1.a := 3\ \{x_2.a = 4\}.$$

If inheritance allows *attribute hiding* in the sense that the attribute a of C_2 can be redeclared in its subclass C_1, even the following Hoare-triple does not generally hold:

$$\{x_1.a = 3\}\ x_2 := x_1\ \{x_2.a = 3\}.$$

Therefore, the following fundamental *backward substitution rule* does not generally hold for OO programs:

$$\{Q[e/le]\}\ le := e\ \{Q\}.$$

In order to allow the use of OO design and programming for component-based software development, rCOS extends the theory of designs in UTP to a theory of OO designs. The theory includes an UTP-based denotational semantics [26, 66], a graph-based operational semantics of OO programs [36] and a refinement calculus [66] of OO designs. We only give a summary of the main ideas and we refer to the publications for technical details, which are of less interest for general readers.

OO Specification. The rCOS OO specification language is defined in [26]. Similar to Java, an OO program P consists of a list *ClassDecls* of class declarations and a main program body *Main*. Each class in *ClassDecls* is of the form:

$$\begin{array}{lll}
\textbf{class}\ M\ [\textbf{extends}\ N] & & \\
 & \textbf{private} & T_{11}\ a_{11} = d_{11}, \ldots, T_{1n_1}\ a_{1n_1} = d_{1n_1}; \\
 & \textbf{protected} & T_{21}\ a_{21} = d_{21}, \ldots, T_{2n_2}\ a_{2n_2} = d_{2n_2}; \\
 & \textbf{public} & T_{31}\ a_{31} = d_{31}, \ldots, T_{3n_3}\ a_{3n_3} = d_{3n_3}; \\
 & \textbf{method} & m_1\ (T_{11}\ x_1; T_{12}\ y_1)\ \{c_1\} \\
 & & \ldots \\
 & & m_\ell\ (T_{\ell 1}\ x_\ell; T_{\ell 2}\ y_\ell)\ \{c_\ell\}
\end{array}$$

Therefore, a class can declare at most one direct superclass using **extends**, some attributes with their types and initial values, and methods with their signatures and body commands. Types include classes and a set of assumed primitive data types such as integers, booleans, characters and strings. The scopes of *visibility* of the attributes are defined by the **private**, **protected** and **public** keywords. We could also have different scopes of visibility for the methods, but we assume all methods are public for simplicity. A method can have a list of input parameters and return parameters with their types. We use return parameters, instead of return types of methods to a) avoid the use of method invocations in expressions so that evaluation of expressions have no side effect, and b) give us the flexibility in specifications that a method can have a number of return values.

The main program body *Main* declares a list *vars* of variables, called the *global variables* with their types and initial values, and a command c. We can thus denote the main program body as a pair $(vars, c)$ in our discussion. One can view the main program body as a class *Main*:

$$\textbf{class}\ Main\ \{\ \textbf{private}\ vars;\ \textbf{method}\ main()\{c\}\ \}$$

A command in a method, including the *main* method, is written in the following syntax:

$$\begin{array}{ll}
\text{expressions:} & e ::= x \mid null \mid this \mid e.a \mid (C)e \mid f(e) \\
\text{assignable expressions:} & le ::= x \mid e.a \\
\text{commands:} & c ::= \textbf{skip} \mid \textbf{chaos} \mid \textbf{var}\ T\ x = e;\ c;\ \textbf{end}\ x \mid \\
 & \qquad c;c \mid c \lhd b \rhd c \mid c \sqcap c \mid b * c \mid \\
 & \qquad e.m(e^*; le) \mid le := e \mid C.new(le)
\end{array}$$

Here, x is a basic type or an object variable and $e.a$ an attribute of e. For the sake of clarity, a simplified presentation for method parameters and variable scope is used; we generally allow lists of expressions as method parameters and lists of variable declarations for the scope operator **var**. Notice that the creation of a new object $C.new(le)$ is a command not an expression. It returns in le the object newly created and plays the same role as $le = new\ C()$ in Java or C++.

Objects, Types and States. An *object* has an identity, a state and a behavior. We use a designated set *REF* to represent object identities. An object also has a runtime type. Thus, we define an object by a triple $o = (r, C, s)$ of its identity r, runtime type C and state s. The state s is a *typed function*

$$s : \mathcal{A}(C) \to \mathcal{O} \cup \mathcal{V},$$

where

– $\mathcal{O}$ is the set of all objects of all classes,
– $\mathcal{V}$ the value space of all the primitive data types,
– $\mathcal{A}(C)$ is the set of names of the attributes of C, including those inherited from all its superclasses, and
– s maps an attribute a to an object or value of the type of a declared in C.

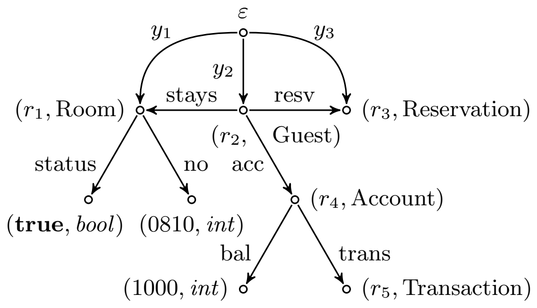
Therefore, an object o has a recursive structure, and can be represented by a rooted-labeled-directed graph, called an *object graph* [36, 66], in which

– the root represents the object labeled by its runtime type,
– each outgoing edge is labeled by an attribute of the object and leads to a node that is either an object or a value, and
– each object node is the root of a subgraph representing that object.

In an object graph, all value nodes are leaves. An object graph can also be represented by a UML object diagram [66], but UML object diagrams do not have the properties of the mathematical structure of rooted-labeled-directed graphs needed for formal reasoning and analysis. Furthermore, the types in an object graph together with the labels for attributes form a *class graph* that is called the *type graph* of the object that the object graph represents [36, 66].

States of Programs. Given an OO program $P = \mathit{ClassDecls} \bullet \mathit{Main}$, a *global state* of P is defined as a mapping $s : \mathit{vars} \to \mathcal{O} \cup \mathcal{V}$ that assigns each variable $x \in \mathit{vars}$ an object or a value depending on the type of x. Taking *Main* as a class, a global state of P is thus an object of *Main* and can be represented as an object graph, called a *global state graph*. During the execution of the main method, the identity of the object representing the state will never be changed, but its state will be modified in each step of the execution. All the global state graphs have the same type graph. The type graph of the program can be statically defined from the class declarations *ClassDecls*. Its UML counterpart is the UML class diagram of the program in which classes have no methods. For example, Fig. 1 is a global state of the accompanied program outline, and its type graph (and the corresponding UML class diagram) is given in Fig. 2.

Global states are enough for defining a UTP-based denotational semantics [26] and a "big step semantics" of the program in which executions of intermediate execution steps and the change of locally declared variables are hidden. To define a small step operational semantics, we need to represent the *stack* of local variable declarations to characterize the execution of **var** $T\ x = x_0$, where T can either



```
program Hotel {
  class Person {
    public Account acc = null;
    method m1(){...}
  }
  class Guest extends Person {
    public
      Room stays = null,
      Reservation resv = null;
    method
      makeReservation() {...}
      checkOut() {...}
  }
  class Account {
    public
      int bal = 0;
      Transaction trans = null;
    method checkBalance() {...}
  }
  class Room {
    public
      int no = 0,
      bool status = false;
    method changeStatus() {...}
  }
  class Transaction {
    ...
  }
  class Main {
    private
      Room y1,
      Guest y2,
      Reservation y3;
    method main() {...}
  }
} // end of program
```

Fig. 1. An example of object graph

be a class or a data type, and x_0 is the declared initial value of x. For this, we extend the notation of global state graphs by introducing edges labeled by a designated symbol \$. The execution of **var** $T\ x = x_0$ from a state graph G adds a new root node n' to G that has an outgoing edge to the root n of G, labeled by \$, and another outgoing edge to x_0, labeled by x. We can understand this as *pushing* a new node on top of G with one outgoing edge labeled by \$ to the root of G and another labeled by x to its initial value. Such a *state graph* contains a \$-path of *scope nodes*, called the *stack*. Executing the command **end** x from such a state graph pops out the root together with its outgoing edges. Figure 3 shows an example of a state graph that characterizes the local scopes below:

$$\textbf{var}\ C_2\ z = o_2, C_3\ x = o_3; \textbf{var}\ C_2\ x = o_2; \textbf{var}\ \mathit{int}\ z = 3, C_1\ y = o_1$$

where o_1, o_2 and o_3 are objects of type C_1, C_2, and C_3 referred to by the variables y, z and x in their scopes, respectively.

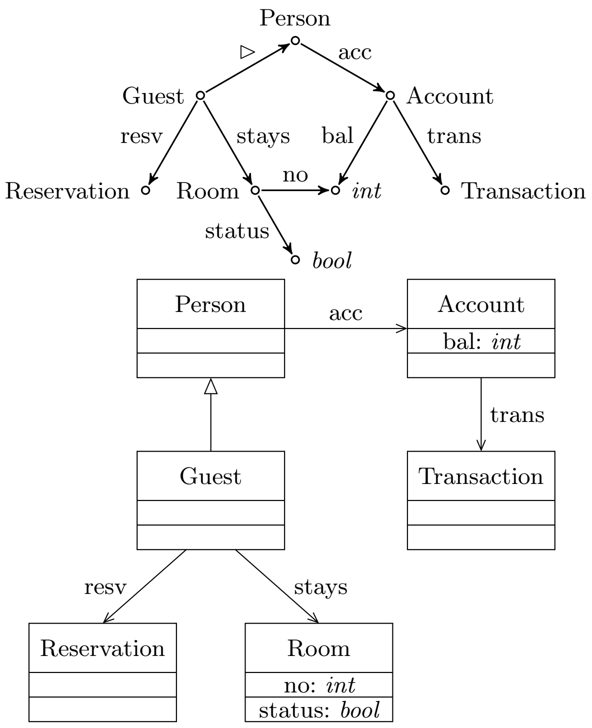


Fig. 2. An example of class graph and diagram

Semantics. We explain the semantics informally. Both a denotational semantics and an operational semantics can be defined by specifying which changes the execution of a command makes on a given state graph. This can be understood with our graph representation of states. Given an initial state graph G

– *assignment*: $le.a := e$ first evaluates e on G as a node n' of G and then swings the a-edge of the target node of le in G to the node n';
– *object-creation*: $C.new(le.a)$ makes an object graph of C according to the initial values of its attributes, such that the root n' is not in G, and then swings the a-edge of the target node of le in G to the node n';
– *method invocation*: $e.m(e_1; le.a)$ first records e, e_1 and le to *this*, the formal value parameter of $m()$ and y^+, respectively, then executes the body of $m()$, returns the value of the formal return parameter of $m()$ to the actual return parameter $y^+.a$ which is the initial $le.a$ that might be changed by the execution, roughly that is

$$\begin{array}{l}\mathbf{var}\ \mathit{this} = e, \mathit{in} = e_1, y^+ = le, \mathit{return};\\ c; y^+.a := \mathit{return};\\ \mathbf{end}\ \mathit{this}, \mathit{in}, y^+, \mathit{return}\end{array}$$

where *in* and *return* are the formal parameters of $m()$.

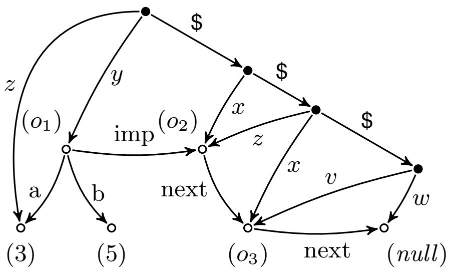


Fig. 3. An example of state graph with local scopes

Then conditional choice, non-deterministic choice and iterative statements can be defined inductively. For a denotational semantics, a partial order has to be defined with that a unique fixed-point of the iterative statements and recursive method invocations can be defined. The theory of denotational semantics is presented in [26] and the graph-based operational semantics is given in [36].

OO Refinement. OO refinement is studied at three levels in rCOS: refinement between whole programs, refinement between class declarations (called *structure refinement*), and refinement between commands. The refinement relation between commands takes exactly the same view as in the previous section about traditional sequential programs, where the execution of a program command is a relation between states. A command c_l is a refinement of a command c_h, denoted by $c_h \sqsubseteq c_l$, if for any given initial state graph G, any possible final state G' of c_l is also a possible final state of c_h. This definition takes care of non-determinism and a refined command is not more non-deterministic than the original command. However, refinement between commands in OO programming only makes sense under the context of a given list of class declarations *ClassDecls*. Under such a given context, some variables and method invocations in a command c might not be defined. In this case, we treat the command to be equivalent to **chaos**, which can be refined by any command under the same context. To compare two commands under different class contexts, we use the extended notation of data refinement and relate the context of c_l to that of c_h by a *class (graph) transformation.*

The combination of class graph transformations and command transformations is illustrated in Fig. 4. It shows that given a class graph transformation ρ from CG to CG_1, we can derive a transformation ρ_o from an instance object graph OG of CG to an instance object graph OG_1 of CG_1, as well as a transformation ρ_c on commands. Then ρ is a class refinement if the diagram commutes for all OG of CG and all commands c.

Definition 4 (OO program refinement). *A program P_l = $ClassDecls_l \bullet Main_l$ is a refinement of a program P_h = $ClassDecls_h \bullet Main_h$, if there is a class graph transformation from the class graph of P_l to that of P_h such that the command of $Main_l$ is a refinement of the command of $Main_h$.*

$$\begin{tikzcd}
CG \arrow[r, "i"] \arrow[d, "\rho"] & OG \arrow[r, "c"] \arrow[d, "\rho_o"] & OG' \arrow[d, "\rho_o"] \\
CG_1 \arrow[r, "i"] & OG_1 \arrow[r, "\rho_c(c)"] & OG'_1
\end{tikzcd}$$

Fig. 4. Class graph transformations and command transformations

In the paper [66], we give a systematic study of the combination of class refinement and command refinement, and develop a graph-based OO refinement calculus. It gives a full formalization of OO program refactoring [23] by a group of simple rules of class graph transformations, including adding classes, attributes, methods, decomposition and composition of classes, promoting methods from subclasses to super classes, from private to protected and then to public. The combination of class graph transformations with command transformations formalizes the design patterns for class responsibility assignments for object-oriented design, including in particular the *expert patterns*, *low coupling* and *high cohesion* patterns [39]. The use of these patterns is an essential practice in OO program design [12].

An essential advantage of OO programming is that classes can be reused in different applications that are implemented in different main methods. Classes can also be extended for application evolution. The classes of an application program are in fact the metamodel of the structure or organization of the application domain in terms of concepts, objects, their relations, and behavior. On the other hand, the main method of the program is the automation of the application business processes, i.e., *use cases*, via the control of the objects' behavior. Of course, different structures provide different functionalities and thus different use cases, the same use case can also be supported by different structures. The structure refinement in rCOS characterizes this fundamental feature of OO programming.

Definition 5 (OO structure refinement). *A list $ClassDecls_l$ of class declarations is a refinement of a list $ClassDecls_h$ of class declarations if for any main method Main, $ClassDecls_h \bullet Main \sqsubseteq ClassDecls_l \bullet Main$.*

This means that a refined list of class declarations has more capacity in providing more and "better" services in the sense that the lower level class declarations may provide additional functionality or may provide more defined functionality with less non-determinism following the classical notion of refinement.

The refinement calculus is proved to be sound and relatively complete in the sense that the rules allow us to transform the class graph of a program to a tree of inheritance, and with the derived transformation rules on the main method, the program can be refined to an equivalent program that only has the main class. Thus each OO program can be transformed to an equivalent procedural program [66].

2.3 Reactive Systems and Reactive Designs

The programs that have been considered so far in this section are sequential and object-oriented programs. For these programs, our semantic definition is only concerned with the relation between the initial and final states and the termination of execution. In general, in a *concurrent* or *reactive* program, a number of components (usually called processes) are running in parallel, each following its own thread of control. However, these processes interact with each other and/or with the environment (in the case of a reactive program) to exchange data and to synchronize their behavior. The termination of the processes and the program as whole is usually not a required property, though the *enabling condition* and *termination* of execution of individual actions are essential for the progress of all processes, i.e., they do not show *livelock* or *deadlock* behavior.

There are mainly two different paradigms of programming interaction and synchronization, shared memory-based programming and message-passing programming. However, there can be programs using both synchronization mechanisms, in distributed systems in which processes on the same node interact through shared variables, and processes on different nodes interact through message passing. We define a general model of *labeled transition systems* for describing the behavior of reactive systems.

Reactive Designs. In general a reactive program can be considered as a set of *atomic actions* programmed in a concurrent programming language. The execution of such an atomic action carries out interactions with the environment and changes of the state of the variables. We give a symbolic name for each atomic action, which will be used to label the state transitions when defining the execution of a reactive program.

The execution of an atomic action changes the current state of the program to another state, just in the way a piece of sequential code does, thus it can be specified as a design $p \vdash R$. However, the execution requires resources that might be occupied by another process or a synchronization condition. The execution is then suspended in a *waiting* state. For allowing the observation of the waiting state, we introduce the new boolean state variables *wait* and *wait′* and define the following lifting function on designs

$$\mathcal{H}(D) \mathrel{\widehat{=}} wait' \lhd wait \rhd D,$$

specifying that the execution cannot proceed in a waiting state. Note that *wait* is not a program variable, and thus cannot be directly changed by a program command. Instead, *wait* allows us to observe waiting states when talking about the semantics of reactive programs. We call a design D a *reactive design* if $\mathcal{H}(D) = D$. Notice that $\mathcal{H}(\mathcal{H}(D)) = \mathcal{H}(D)$.

Theorem 3 (Reactive design). *The domain of reactive designs has the following closure properties:*

$$\begin{aligned}\mathcal{H}(D_1 \vee D_2) &= \mathcal{H}(D_1) \vee \mathcal{H}(D_2),\\ \mathcal{H}(D_1; D_2) &= \mathcal{H}(D_1); \mathcal{H}(D_2),\\ \mathcal{H}(D_1 \lhd b \rhd D_2) &= \mathcal{H}(D_1) \lhd b \rhd \mathcal{H}(D_2).\end{aligned}$$

Given a reactive design D and a state predicate g, we call $g \,\&\, D$ a *guarded design* and its meaning is defined by

$$g \,\&\, D \mathrel{\widehat{=}} D \lhd g \rhd (\mathit{true} \vdash \mathit{wait}').$$

Theorem 4. *If D is a reactive design, so is $g \,\&\, D$.*

For non-reactive designs $p \vdash R$, we use the notation $g \,\&\, (p \vdash R)$ to denote the guarded design $g \,\&\, \mathcal{H}(p \vdash R)$, where it can be proved $\mathcal{H}(p \vdash R) = (\mathit{wait} \vee p) \vdash (\mathit{wait}' \lhd \mathit{wait} \rhd R)$. This guarded design specifies that if the guard condition g holds, the execution of design proceeds from non-waiting state, otherwise the execution is suspended. It is easy to prove that a guarded design is a reactive design.

Theorem 5 (Guarded design). *For guarded designs, we have*

$$\begin{aligned}g_1 \,\&\, D_1 \lhd b \rhd g_2 \,\&\, D_2 &= (g_1 \lhd b \rhd g_2) \,\&\, (D_1 \lhd b \rhd D_2),\\ g_1 \,\&\, D_1; g_2 \,\&\, D_2 &= g_1 \,\&\, (D_1; g_2 \,\&\, D_2),\\ g \,\&\, D_1 \vee g \,\&\, D_2 &= g \,\&\, (D_1 \vee D_2),\\ g \,\&\, D_1; D_2 &= g \,\&\, (D_1; D_2).\end{aligned}$$

A concurrent program P is a set of atomic actions, and each action is a *guarded command* in the following syntax:

$$c ::= x := e \mid c; c \mid c \lhd b \rhd c \mid c \,\Box\, c \mid g \,\&\, c \mid b * c \tag{4}$$

Note that $x := e$ is interpreted as command guarded by *true*. The semantics of the commands is defined inductively by

$$\begin{aligned}x := e &\mathrel{\widehat{=}} \mathcal{H}(\mathit{true} \vdash x' = e \textstyle\bigwedge_{y \in \alpha, y \not\equiv x} y' = y)\\ g \,\&\, c &\mathrel{\widehat{=}} c \lhd g \rhd (\mathit{true} \vdash \mathit{wait}')\\ g_1 \,\&\, c_1 \,\Box \cdots \Box\, g_n \,\&\, c_n &\mathrel{\widehat{=}} (g_1 \vee \cdots \vee g_n) \,\&\, (g_1 \wedge c_1 \vee \cdots \vee g_n \wedge c_n)\end{aligned}$$

and for all other cases as defined in equation (3) for the sequential case. The semantics and reasoning of concurrent programs written in such a powerful language are quite complicated. The semantics of an atomic action does not generally equal to a guarded design of the form $g \,\&\, p \vdash R$. This imposes difficulty to separate the design of the synchronization conditions, i.e., the guards, from the design of the data functionality. Therefore, most concurrent programming languages only allow guarded commands of the form $g \,\&\, c$ such that no guards are in c anymore. A set of such atomic actions can also be represented as a Back's *action system* [3], a UNITY program [9] and a TLA specification [37].

Labeled State Transition Systems. Labeled transition systems are often used to describe the behavior of reactive systems, and we will use them in the following sections when defining the semantics of components. Hence, the remaining part of this section deals with basic definitions and theorems about labeled transition systems. Intuitively, states are defined by the values of a set of variables including both data variables and variables for the flow of control, which we do not distinguish here. Labels represent events of execution of actions that can be *internal* events or events *observable* by the environments, i.e., interaction events.

Definition 6 (Labeled transition system). *A* **labeled transition system** *is a tuple*

$$S = \langle var, init, \Omega, \Lambda \rangle,$$

where

- *var is the set of typed variables (not including ok and $wait$), denoted $S.var$, we define Σ_{var} to be the set of states over $var \cup \{ok, wait\}$,*
- *$init$ is the initial condition defining the allowable initial states, denoted by $S.init$, and*
- *Ω and Λ are two disjoint sets of named atomic actions, called observable and internal actions, respectively; actions are of the form $a\{c\}$ consisting of a name a and a guarded command c as defined in Syntax (4). Observable actions are also called* interface actions.

In an action $a\{c\}$, we call c the *body* of a. For $\Gamma = \Omega \cup \Lambda$ and for two states s and s' in Σ_{var},

- an action $a \in \Gamma$ is said to be **enabled** at s if for the body c of a the implication $c[s(x)/x] \Rightarrow \neg wait'$ holds, and **disabled** otherwise.
- a state s is a **divergence state** if ok is $false$ and a **deadlock state** if $wait = true$.
- we define $\rightarrow \subseteq \Sigma_{var} \times \{a | a\{c\} \in \Gamma\} \times \Sigma_{var}$ as the state transition relation such that $s \xrightarrow{a} s'$ is a transition of S, if a is enabled at s and s' is a post-state of the body c of action a.

Notice that this is a general definition of labeled transition systems that includes both finite and infinite transition systems, closed concurrent systems in which processes share variables (when all actions are internal), and I/O automata. Further, it models both data rich models in which a state contains values of data variables, and symbolic state machines in which a state is a symbol represents an abstract state of a class of programs. In later sections, we will see the symbols for labeling the actions can also be interpreted as a combination of input events triggering a set of possible sequences of output events.

Definition 7 (Execution, observable execution and stable state). *Given a labeled transition system S,*

1. *an* **execution** *of S is a sequence of transitions $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \cdots \xrightarrow{a_n} s_n$ of S, where $n \geq 0$ and s_i $(0 \leq i \leq n)$ are states over $var \cup \{ok, wait\}$ such that s_0 is an initial state of S.*
2. *a state s is said to be* **unstable** *if there exists an internal action enabled in s. A state that is not unstable is called a* **stable state**.
3. *an* **observable execution** *of S is a sequence of external transitions*

$$s_0 \overset{a_1}{\Longrightarrow} s_1 \overset{a_2}{\Longrightarrow} \cdots \overset{a_n}{\Longrightarrow} s_n$$

where all $a_i \in \Omega$ for $i = 1, \ldots, n$, and $s \overset{a}{\Longrightarrow} s'$ if s and s' there exist internal actions $\tau_1, \ldots, \tau_{k+\ell}$ as well as states t_j for $k, \ell \geq 0$ such that

$$s \xrightarrow{\tau_1} \cdots \xrightarrow{\tau_k} t_k \xrightarrow{a} \cdots \xrightarrow{\tau_{k+\ell}} s'.$$

Notice that the executions (and observable executions) defined above include chaotic executions in which divergence states may occur. Therefore, we give the semantic definitions for transitions systems below following the ideas of *failure-divergence semantics* of CSP.

Definition 8 (Execution semantics). *Let $S = \langle var, init, \Omega, \Lambda\rangle$ be a transition system. The* execution semantics *of S is defined by a pair $(\mathcal{ED}(S), \mathcal{EF}(S))$ of* execution divergences *and* execution failures*, where*

1. *A divergence execution in $\mathcal{ED}(S)$ is a finite observable execution sequence of S*

$$s_0 \overset{a_1}{\Longrightarrow} s_1 \overset{a_2}{\Longrightarrow} \cdots \overset{a_n}{\Longrightarrow} s_n$$

where there exists an divergence state s_k, $k \leq n$. Notice that if s_k is a divergence state, each s_j with $k \leq j \leq n$ is also a divergence state.

2. *The set $\mathcal{EF}(S)$ contains all the pairs (σ, X) where σ is a finite observable execution sequence of S and $X \subseteq \Omega$ such that one of the following conditions hold*
   (a) σ is empty, denoted by ε, and there exists an allowable initial state s_0 such that a is disabled at s_0 for any $a \in X$ or s_0 is unstable and X can be any set,
   (b) $\sigma \in \mathcal{ED}(S)$ and X can be any subset of Ω, i.e., any interaction with the environment can be refused,
   (c) $\sigma = s_0 \overset{a_1}{\Longrightarrow} \cdots \overset{a_k}{\Longrightarrow} s_k$ and for any s in the sequence, $s(ok) = true$ and $s_k(wait) = false$, and each $a \in X$ is disabled at s_k, or s_k is unstable and X can be any set.

The semantics takes both traces and data into account. The component X of $(\sigma, X) \in \mathcal{EF}(S)$ is called a set of *refusals* after the execution sequence tr. We call the subset $ExTrace(S) = \{\sigma \mid (\sigma, \emptyset) \in \mathcal{EF}(S)\}$ the *normal execution traces*, or simply *execution traces*.

When interaction behavior and properties are the main interest, we can omit the states from the sequences and define the *interaction divergences* $\mathcal{ID}(S)$ and *interaction failures* $\mathcal{IF}(S)$ as

$$\mathcal{ID}(S) = \{a_1 \dots a_n \mid s_0 \overset{a_1}{\Longrightarrow} s_1 \overset{a_2}{\Longrightarrow} \cdots \overset{a_n}{\Longrightarrow} s_n \in \mathcal{ED}(S)\}$$
$$\mathcal{IF}(S) = \{(a_1 \dots a_n, X) \mid (s_0 \overset{a_1}{\Longrightarrow} s_1 \overset{a_2}{\Longrightarrow} \cdots \overset{a_n}{\Longrightarrow} s_n, X) \in \mathcal{EF}(S)\}$$

We call the set $\mathcal{T}(S) = \{\sigma \mid (tr, \emptyset) \in \mathcal{IF}(S)\}$ the *normal interaction traces*, or simply *traces*. Also, when interaction is the sole interest, abstraction would be applied to the states so as to generate transition systems with *symbolic states* for the flow of control. Most existing modeling theories and verification techniques work effectively on transition systems with finite number of states, i.e., finite state systems.

Definition 9 (Refinement of reactive programs). *Let*

$$S_l = \langle var, init_l, \Omega_l, \Lambda_l \rangle \quad and \quad S_h = \langle var, init_h, \Omega_h, \Lambda_h \rangle$$

be transition systems. S_l *is a* **refinement** *of* S_h*, denoted by* $S_h \sqsubseteq S_l$*, if* $\mathcal{ED}(S_l) \subseteq \mathcal{ED}(S_h)$ *and* $\mathcal{EF}(S_l) \subseteq \mathcal{EF}(S_h)$*, meaning that* S_l *is not more likely to diverge or deadlock when interacting with the environment through the interface actions* Ω*.*

Notice that we, for simplicity, assume that S_l and S_h have the same set of variables. When they have different variables, the refinement relation can be defined through a state mapping (called refinement mapping in TLA [37]).

A labeled transition system is a general computational model for reactive programs developed in different technologies. Thus, the definition of refinement will lead to a refinement calculus when a modeling notation of reactive programs is defined that includes models of primitive components and their compositions. We will discuss this later when the rCOS notation is introduced, but the discussion will not be in great depth as we focus on defining the meaning of component-based software architecture. On the other hand, the following theorem provides a verification technique for checking refinement of transition systems that is similar to the relation of simulation of transition systems, but extended with data states.

Theorem 6 (Refinement by simulation). *For two transition systems* S_h *and* S_l *such that they have the same set of variables,*

- *let* $guard_h(a)$ *and* $guard_l(a)$ *be the enabling conditions, i.e., the guards* g *for an action* a *with body* $g \,\&\, c$ *in* S_h *and* S_l*, respectively,*
- $next_h(a)$ *and* $next_l(a)$ *are the designs, i.e., predicates in the form of* $p \vdash R$*, specifying the state transition relations defined by the body of an action* $a\{g \,\&\, (p \vdash R)\}$ *in* S_h *and* S_l*, respectively,*
- $g(\Omega_h)$*,* $g(\Lambda_h)$*,* $g(\Omega_l)$ *and* $g(\Lambda_l)$ *are the disjunctions of the guards of the interface actions and invisible actions of the programs* S_h *and* S_l*, respectively,*
- $inext(S_h) = \bigvee_{a \in \Lambda_h} guard_h(a) \wedge next_h(a)$ *the state transitions defined by the invisible actions of* S_h*, and*
- $inext(S_l)$ *analogously defined as* $inext(S_h)$ *above.*

We have $S_h \sqsubseteq S_l$ if the following conditions holds

1. *$S_l.init \Rightarrow S_h.init$, i.e., the initial condition of S_h is preserved by S_l,*
2. *for each $a \in \Omega_l$, $a \in \Omega_h$ and $guard_l(a) \iff guard_h(a)$,*
3. *for each $a \in \Omega_l$, $a \in \Omega_h$ and $next_h(a) \sqsubseteq next_l(a)$, and*
4. *$\neg g(\Omega_h) \implies (g(\Lambda_h) \iff g(\Lambda_l)) \wedge (inext(S_l) \implies inext(S_h))$, i.e., any possible internal action of S_l in an unstable state would be a transition allowable by an internal action of S_h.*

When $inext(S_h) \sqsubseteq inext(S_l)$, the fourth condition can be weakened to

$$\neg g(\Omega_h) \implies (g(\Lambda_h) \iff g(\Lambda_l))$$

In summary, the first condition ensures the allowable initial states of S_l are allowable for S_h; the second ensures S_l is not more likely to deadlock; the third guarantees that S_l is not more non-deterministic, thus not more likely to diverge, than S_h, and the fourth condition ensures any refining of the internal action in S_l should not introduce more deadlock because of removing internal transitions from unstable states. Notice that we cannot weaken the guards of the actions in a refinement as otherwise some safety properties can be violated.

This semantics extends and unifies the theories of refinement of closed concurrent programs with shared variables in [3, 9, 37, 44] and failure-divergence refinement of CSP [57]. However, the properties of this unified semantics still have to be formally worked out in more detail.

Design and verification of reactive programs are challenging and the scalability of the techniques and tools is fundamental. The key to scalability is compositionality and reuse of design, proofs and verification algorithms. Decomposition of a concurrent program leads to the notion of reactive programs, that we model as components in rCOS. The rCOS component model is presented in the following sections.

3 Model of Primitive Closed Components

The aim of this chapter is to develop a unified model of architecture of components, that are *passive service components* (simply called *components*) and *active coordinating components* (simply referred to as *processes*). This is the first decision that we make for *separation of concerns*. The reason is that components and processes are different in nature, and they play different roles in composing and coordinating services to form larger components. Components maintain and manage data to provide services, whereas processes coordinate and orchestrate services in business processes and workflows. Thus, they exhibit simpler semantic behaviors than "hybrid" components that can have both passive and active behaviors when interacting with the environment. However, as a semantic theory, we develop a unified semantic model for all kinds of architectural components - the passive, active and the general hybrid components. We do this step by step, starting with the passive components, then the active process and finally we will define compositions of components that produces general components with both passive and active behavior. We start in this section with the simplest kind of components - primitive closed components. They are passive.

A closed and passive component on one hand interacts with the environment (users/actors) to provide services and on the other hand carries out data processing and computation in response to those services. Thus, the model of a component consists of the *types* of the data, i.e., the program variables, of the component, the *functionality* of the operations on the data when providing services, and the *protocol* of the interactions in which the component interacts with the environment. The design of a component evolves from the techniques applied during the design process, i.e., decomposing, analyzing, and integrating different *viewpoints* to form a correctly functioning whole component, providing the services required by the environment. The model of a component is separated into a number of related models of different viewpoints, including static structure, static data functionality, interaction protocol, and dynamic control behavior. This separation of design concerns of these viewpoints is crucial to a) control the complexity of the models, and b) allow the appropriate use of different techniques and tools for modeling, analysis, design, and verification.

It is important to note that the types of program data are not regarded as a difficult design issue anymore. However, when *object-oriented programming* is used in the design and implementation of a component-based software system, the types, i.e., the classes of objects become complicated and their design is much more tightly coupled with the design of the functionality of a component. The rCOS method presents a combination of OO technology and component-based technology in which local data functionality is modeled with the unified theory of sequential programming, as discussed in the previous section.

3.1 Specification Notation for Primitive Closed Components

To develop tool support for a formal method, there is a need for a specification notation. In rCOS, the specification notation is actually a graphical input notation implemented in a tool, called *the rCOS modeler*.[3] However, in this chapter the specification notation is introduced incrementally so as to show how architectural components, their operations and semantics can be defined and used in examples. We first start with the simplest building blocks[4] in component software, which we call *primitive closed components*. Closed components *provide services* to the environment but they do not *require services* from other components to deliver the services. They are passive as they wait for the environment to call their provided services, having no autonomous actions to interact with the environment. Furthermore, being primitive components, they do not have internal autonomous actions that result from interaction among sub-components. We use the notation illustrated in Fig. 5 to specify primitive closed components, which is explained as follows.

[3] `http://rcos.iist.unu.edu`

[4] In the sense of concepts and properties rather than size of software, e.g., measured by number of lines of code.

```
component K {
   T x = c; // initial state of component
   provided interface I { // provided methods
      m1(parameters) { g1 & c1 /* functionality definition */ };
      m2(parameters) { g2 & c2 /* functionality definition */ };
      ...
      m(parameters) { g & c /* functionality definition */ };
   };
   internal interface { // locally defined methods
      n1(parameters) { h1 & d1 /* functionality definition */ };
      n2(parameters) { h2 & d2 /* functionality definition */ };
      ...
      n(parameters) { h & d /* functionality definition */ };
   }
   class C1{...}; class C2{...}; ... // used in the above specification
}
```

Fig. 5. Format of rCOS closed components

Interfaces of Components. The *provided interface* declares a list of methods or services that can be invoked or requested by clients. The interface also allows declarations of state variables. A closed component only provides services, and thus, it has only a provided interface and optionally an *internal interface*, which declares private methods. Private methods can only be called by provided or private methods of the same component.

Access Control and Data Functionality. The control to the access and the data functionality of a method m, in a provided or internal interface, is defined by a combination of a guard g and a command c in the form of a guarded command $g \,\&\, c$.

The components that we will discuss in the rest of this section are all primitive closed components. This definition emphasizes on the *interface* of the provided services. The interface supports input and output identifications, data variables, and the functional description defined by the bodies of the interface methods. On the other hand, the guards of the methods are used to ensure that services are provided in the right order.

Based on the theory of guarded designs presented in Sect. 2, we assume that in a closed component the access control and data functionality of each provided interface method m is defined by a guarded design $g \,\&\, D$. For a component K, we use $K.pIF$ to denote the provided interface of K, $K.iIF$ the internal interface of K, $K.var$ the variables of K, $K.init$ the set of initial states of K. Furthermore, we use $guard(m)$ and $body(m)$ to denote the guard g and the body D of m, respectively. For the sake of simplicity but without loosing theoretical generality, we only consider methods with at most one input parameter and at most one return parameter.

We define the behavior of component K by the transition relation of $\underline{K}$ defined in the next subsection.

3.2 Labeled Transition Systems of Primitive Closed Components

We now show that each primitive closed component specified using the rCOS notation can be defined by a labeled transition system defined in Sect. 2.3. To this end, for each method definition $m(T_1\ x; T_2\ y)\{c\}$, we define the following set of events

$$\omega(m) = \{m(u)\{c[u/x, v/y]\} \mid u \in T_1\}.$$

We further define $\Omega(K) = \bigcup_{m \in K.pIF} \omega(m)$. Here, there is a quite subtle reason why the return parameter is not included in the events. It is because that

- returning a value is an "output" event to the environment and the choice of a return value is decided by the component itself, instead of the environment,
- we assume that the guards of provided methods do not depend on their return values,
- we assume a run to complete semantics, thus the termination of a method invocation does not depend on the output values of the methods, and
- most significantly, it is the data functionality design, i.e. design $p \vdash R$, of a method, that determines the range of non-deterministic choices of the return values of an invocation for a given input parameter, thus refining the design will reduce the range of non-determinism.

Definition 10 (Transition system of primitive closed component). *For a primitive closed component K, we define the transition system*

$$\underline{K} = \langle K.var, K.init, \Omega(K), \emptyset \rangle$$

A transition $s \xrightarrow{m(u)} s'$ *of $\underline{K}$ is an execution of the invocation $m(u)$ if the following conditions hold,*

1. *the state space of $\underline{K}$ is the states over $K.var$, $\Sigma_{K.var}$,*
2. *the initial states of $\underline{K}$ are the same initial states of K,*
3. *s and s' are states of K,*
4. *$m(u) \in \Omega(K)$ and it is an invocation of a provided method m with in input value u,*
5. *$s \oplus u$ satisfies $guard(m)$, i.e., m is enabled in state s for the input u (note that we identify the value u with the corresponding state assigning values to inputs $u = u(in)$), and there exists a state v of the output parameter y of m*
6. *$(s \oplus u, s' \oplus v) \in body(m)$.*

We omit the empty set of internal actions and denote the transition system of K by $= \langle K.var, K.init, \Omega(K) \rangle$. A step of state transition is defined by the design of the method body when the guard holds in the starting state s For transition $t = s \xrightarrow{m(u)} s'$, we use $pre.t$, $post.t$ and $event.t$ to denote the pre-state s, the post-state s' and the event $m(u)$, respectively.

Definition 11 (Failure-divergence semantics of components). *The* **execution failure-divergence semantics** $\langle \mathcal{ED}(K), \mathcal{EF}(S)\rangle$ *(or the* **interaction failure-divergence semantics** $\langle \mathcal{ID}(K), \mathcal{IF}(S)\rangle$*) of a component* K *is defined by the semantics of the corresponding labeled transition system, i.e., by the execution failure-divergence semantics* $\langle \mathcal{ED}(\underline{K}), \mathcal{EF}(\underline{K})\rangle$ *(or the interaction failure-divergence semantics* $\langle \mathcal{ID}(\underline{K}), \mathcal{IF}(\underline{K})\rangle$*).*

The traces $\mathcal{T}(K)$ of K are also defined by the traces of the corresponding transition system: $\mathcal{T}(K) \mathrel{\widehat{=}} \mathcal{T}(\underline{K})$.

Example 1. To illustrate a reactive component using guarded commands, we give an example of a component model below describing the behavior of a memory that a processor can interact with to write and read the value of the memory. It provides two methods for writing a value to and reading the content out of the memory cell of type Z, requiring that the first operation has to be a write operation.

```
component M {
   provided interface MIF {
      Z d;
      bool start = false;
      W(Z v) { true & (d := v; start := true) }
      R(; Z v) { start & (v := d) }
   }
}
```

Relation to Traditional Theories of Programming. We would like to make the following important notes on the expressiveness of this model by relating it to traditional theories.

1. This model is very much similar to the model of Temporal Logic of Actions (TLA) for concurrent programs [38]. However, "actions" in TLA are autonomous and models interact through shared variables. Here, a component is a passive entity and it interacts with the environment through method invocations. Another significant difference between rCOS and TLA is that rCOS combines state-based modeling of data changes and event-based description of interaction protocols or behavior.
2. In the same way as to TLA, the model of components in rCOS is related to Back's *action systems* [3] that extends Dijkstra's *guarded commands language* [17] to concurrent programming.
3. The model of components here is similar to I/O automata [48]. However, the guards of methods in general may depend on input parameters, as well as state variables of the component. This implies that a component may be an I/O automata with infinite number of states. The I/O automata used for verification, model checking in particular, are finite state automata and the states are abstract symbolic states. The guards of transitions are also encoded in the symbolic states such that in some states of an automaton, transitions are enabled or disabled.

4. Similar to the relation with I/O automata, the rCOS model of components combines data state changes with event-based interaction behavior. The latter can be specified in CSP [28, 57]. Indeed, failure-divergence semantics and the traces of a component K are directly influenced by the concepts and definitions in CSP. However, an event $m(u)$ in rCOS is an abstraction of the *extended rendezvous* for the synchronizations of receiving an invocation to m and returning the completion of the execution of m. This assumes a *run to complete semantics for method invocations.* For the relation between I/O automata and process algebras, we refer to the paper by Vaandrager [61].
5. Other formalisms like, e.g. CSP-OZ [22, 30], also combine state and event-based interaction models in a similar way. These approaches and also similar combinations like Circus [64] share the idea of rCOS that different formal techniques are necessary to cope with the complexity of most non-trivial applications. Contrary to rCOS, they promote the combination of fixed existing formal languages, whereas the spirit of rCOS is to provide a general semantic framework and leaving the choice of the concrete applied formalisms to the engineers.

The above relations show that the rCOS model of components unifies the semantics models of data, data functionality of each step of interaction, and event-based interaction behavior. However, the purpose of the unification is not to "mix them together" for the expressive power. Instead, the unification is for their consistent integration and the separation of the treatments of the different concerns. Therefore, rCOS promotes the ideas of *Unifying Theories of Programming* [8, 29] for *Separation of Concerns*, instead of extending a notation to increase expressive power.

3.3 Component Contracts and Publications

We continue with defining necessary constructs for component-based design, i.e., contracts, provided protocols, and publications.

Definition 12 (Contract). *A* **component contract** *C is just like a primitive component, but the body of each method $m \in C.pIF$ is a guarded design g_m & $(p_m \vdash R_m)$.*

So each closed component K is semantically equivalent to a contract. Contracts are thus an important notion for the requirement specification and verification of the correct design and implementation through refinements. They can be easily modeled by a state machine, which is the vehicle of model checking. The contract of the component M of Example 1 on page 27 is given as follows.

```
component M {
   provided interface MIF {
      Z d; bool start = false;
      W(Z v) { true & ({d,start}:true ⊢ d' = v ∧ start' = true) }
      R(; Z v) { start & ({v}: true ⊢ v' = d) }
   }
}
```

Notice that in both the component M of Example 1 and its contract, the state variable *start* is a protocol control variable.

Clearly, for each component contract C, the labeled actions in the corresponding transition system $\underline{C}$ are all of the form $m(T_1\ x; T_2\ y)\{g\ \&\ (p \vdash R)\}$. Notice that in general a method of the provided interface can be non-deterministic, especially at a high level abstraction. Some of the traces are non-deterministic in a way that a client can still get blocked, even if it interacts with K following such a trace from the provided interface. Therefore, $\mathcal{T}(K)$ cannot be used as a description of the provided protocol of the component, for third party composition, because a protocol is commonly assumed to ensure non-blocking behavior.

Definition 13 (Input-deterministic trace and protocol). *We call a trace* $tr = a_1 \cdots a_n$ *of a component transition system* $\underline{K}$ **input-deterministic** *or* **non-blockable** *if for any of its prefixes* $pref = a_1 \cdots a_k$*, there does not exist a set* X *of provided events of* K *such that* $a_{k+1} \in X$ *and* $(pref, X) \in \mathcal{IF}(K)$*. And for a closed component* K*, we call the set of its input deterministic traces the* **provided protocol** *of* K*, and we denote it by* $\mathcal{PP}(K)$ *(and also* $\mathcal{PP}(\underline{K})$*).*

The notion of contract is a rather "operational" model in the sense that the behavior is defined through the enabledness of a method at a state and the transition to the next possible state. This model has its advantage in supporting model checking verification techniques. However, for such an operational model with its non-determinism it is not easy to support third party usage and composition. A more denotational or global behavioral model would be more appropriate. Hence, we define the notion of protocols of components and publications of components below. From the behavioral semantics of a contract C defined by Definition 11, we obtain the following model interface behavior.

Definition 14 (Publication). *A* **component publication** B *is similar to a contract and consists of the following sections of declarations,*

– *its provided interface* $B.pIF$*,*
– *variables* $B.var$ *and initial states* $B.init$*,*
– *the data functionality specification* $m(T_1\ x; T_2\ y)\{p \vdash R\}$*, and*
– *the provided protocol* $B.pProt$ *that is a set of traces.*

A contract C can be transformed into a component publication by embedding the guards of methods into the protocol. That is, the component publication for C is obtained by using the set of non-blockable traces of $\underline{C}$ as provided protocol $\mathcal{PP}(C)$ and by removing the guards of interface methods. For the same reason, the state variables that are only used in the flow of interaction control, such as *start* in the memory component M in Example 1, can also be abstracted away from the publication. The protocol can be specified in a formal notation. This includes formal languages, such as regular expressions or a restricted version of process algebra such as CSP without hiding and internal choice. Publications are declarative, while contracts are operational. Thus, publications are suitable for specifying components in system synthesis.

Example 2. Both the methods W and R of the interface of M in Example 1 are deterministic. Thus, M is input-deterministic and we have

$$\mathcal{PP}(M) = \mathcal{T}(M) = ?W(\{?W, ?R\}^*)$$

Here we adopt the convention to use a question mark to prefix an input service request event, i.e., a method invocation, in order to differentiate it from a calling out event in the required interface of an *open component*, which we will define later. Also, we have omitted the actual parameters in the method invocations, and $?W$ for example represents all possible $?W(a)$ for $a \in Z$. Thus, the following specification is a publication of the memory component M of Example 1.

```
component M {
  provided interface MIF {
    Z d;
    W(Z v) { {d}: true ⊢ d' = v }
    R(; Z v) { {v}: true ⊢ v' = d }
    protocol { ?W({?W,?R})* }
  }
}
```

In the example, we used a regular expression to describe the provided protocol. However, regular expressions have limited expressive power and can only express languages of finite state automata. Languages like CSP can be used for more general protocols.

For the rest of the chapter, we use the programming notation defined in Sect. 2 in place of the designs that define its semantics. We use the notion "component" also for a "contract" and a "publication", as they are specifications of components at different levels of abstractions and for different purposes.

3.4 Refinement between Closed Components

Refinement between two components K_h and K_l, denoted by $K_h \sqsubseteq K_l$, compares the services that they provide to the clients. However, this relation is naturally defined by the refinement relation $\underline{K_h} \sqsubseteq \underline{K_l}$ of their labeled transitions systems. Also, as a specialized form of Theorem 6, we have the following theorem for the refinement relation between two primitive closed components.

Theorem 7. *If* $K_h \sqsubseteq K_l$*,* $\mathcal{PP}(K_h) \subseteq \mathcal{PP}(K_l)$*.*

Proof. The proof is given by induction on the length of traces. From an initial state s_0, e is non-blockable in K_h only if e is enabled in all the possible initial states of K_h. Hence, if e is non-blockable in K_h, so is it in K_l. Assume the theorem holds for all traces of length no longer than $k \geq 1$. If a trace $tr = e_1 \ldots e_k e_{k+1}$ is not blockable in K_h, all its prefixes are non-blockable in K_h, thus so are they in K_l. If tr is blockable in K_l, then there is an X such that $e_{k+1} \in X$ and $(e_1 \ldots e_k, X) \in \mathcal{IF}(K_l)$. Because $K_h \sqsubseteq K_l$, $\mathcal{IF}(K_l) \subseteq \mathcal{IF}(K_h)$, thus $(e_1 \ldots e_k, X) \in \mathcal{IF}(K_h)$. This is impossible because tr is not blockable in S_h. Hence, we have $tr \in \mathcal{PP}(K_l)$.

Thus, a refined component provides more deterministic services to the environment, because protocols represent executions for which there is no internal non-determinism leading to deadlocks.

The result of Theorem 7 is noteworthy, because the subset relation is reversed compared to the usual subset relation defining refinement; for instance, we have $\mathcal{IF}(K_l) \subseteq \mathcal{IF}(K_h)$ and $\mathcal{T}(K_l) \subseteq \mathcal{T}(K_h)$, but $\mathcal{PP}(K_h) \subseteq \mathcal{PP}(K_l)$. However, a bit of thought reveals that this actually makes sense, because removal of failures leads to potentially more protocols. For traces this is a bit more surprising, but in failure-divergence semantics the traces are derived from failures, so they are not independent. This also leads to the fact that the correctness of the theorem actually depends on the divergences: the theorem cannot hold in the stable-failures model and the traces model, because both have a top element regarding the refinement order. For both of these top elements (the terminating process for the trace model and the divergent process for the stable-failures model) the set of protocols is empty. For this reason, we use an extended version of the stable-failures semantics in Sect. 7.

The semantic definition of refinement of components (or contracts) by Definition 2 does not directly support to verify that one component M_l refines another M_h. To solve this problem, we have the following theorem.

Theorem 8. *Let C_l and C_h be two contracts such that $C_l.pIF = C_h.pIF$, (simply denoted as pIF). $M_h \sqsubseteq M_l$ if there is a total mapping from the states over $C_l.var$ to the states over $C_h.var$, $\rho : C_l.var \longmapsto C_h.var$, that can be written as a design with variables in $C_l.var$ and $C_h.var'$ such that the following conditions hold.*

1. *Mapping ρ preserves initial states, i.e., $\rho(C_l.init) \subseteq C_h.init$.*
2. *No guards of the methods of C_h are weakened — undermines safety, or strengthened — introduces likelihood of deadlock, i.e., $\rho \Rightarrow (guard_l(m) \Leftrightarrow guard_h(m)')$ for all $m \in pIF$, where $guard_h(m)'$ is the predicate obtained from $guard_h(m)$ with all its variables replaced by their primed versions,*
3. *The data functionality of each method in C_l refines the data functionality of the corresponding method in C_h, i.e., for all $m \in pIF$,*

$$\rho; body_h(m) \sqsubseteq body_l(m); \rho.$$

The need for the mapping to be total is to ensure that any state in the refined component C_l implements a state in the "abstract contract" C_h. With the upward refinement mapping ρ from the states of C_l at the lower level of abstraction to the states of C_h at a higher level of abstraction, the refinement relation is also called an *upward simulation* and it is denoted by $C_l \preceq_{up} C_h$. Similarly, we have a theorem about *downward simulations*, which are denoted by $C_l \preceq_{down} C_h$.

Theorem 9. *Let C_l and C_h be two contracts. $C_h \sqsubseteq C_l$ if there is a total mapping from the states over $C_h.var$ to the states over $C_l.var$, $\rho : C_h.var \longmapsto C_l.var$, that can be written as a design with variables in $C_l.var'$ and $C_h.var$ such that the following conditions hold.*

1. *Mapping ρ preserves initial states, i.e., $C_l.init \subseteq \rho(C_h.init)$.*
2. *No guards of the methods of C_h are weakened — undermines safety, or strengthened — introduces likelihood of deadlock, i.e., $\rho \Rightarrow (guard_l(m)' \Leftrightarrow guard_h(m))$ for all $m \in pIF$, and*
3. *The data functionality of each method in C_l refines the data functionality of the corresponding method in C_h, i.e., for all $m \in pIF$,*

$$body_h(m); \rho \sqsubseteq \rho; body_l(m).$$

The following theorem shows the completeness of the simulation techniques for proving refinement between components.

Theorem 10. *$C_h \sqsubseteq C_l$ if and only if there exists a contract C such that*

$$C_l \preceq_{up} C \preceq_{down} C_h.$$

The proofs and details of the discussion about the importance of the above theorems can be found in [10].

3.5 Separation of Concerns

A component can be modeled in the specification notation defined in Fig. 5 at different levels of abstraction, and the relation between different levels of abstraction is expressed by refinement. The semantics theory of guarded commands leads to the model of contracts, and a *failure-divergence* semantic model of components. Contracts serve as a requirement model of components that can directly be represented as automata or labeled transition systems. Contracts are also used for correct by construction of components with the refinement calculus.

From the model of contracts and their failure-divergence semantics, we derive the model of publications. The publication of a component eases the usage of a component, including the usage through the interface and the composition with other components to form new software components. For this, a publication has to be faithful with respect to the specification of the component by its contract.

Definition 15. *Let B be a publication and C be a contract such that $B.pIF = C.pIF$, $B.var = C.var$ and $B.init \subseteq C.init$. B is* **faithful** *to C if $B.pProt \subseteq \mathcal{PP}(C)$ and $body_C(m) \sqsubseteq body_B(m)$.*

The faithfulness of a publications states that each method provides the functionality as specified in the contract and all traces published are acceptable traces of the component specified by the contract. Now we have the following *theorem of separation of concerns.*

Theorem 11. *We can separate the analysis, design and verification of the data functionality and the interaction protocol of a component as follows.*

1. *If* $B_1 = (pIF, X, X_0, \Psi, pProt_1)$ *and* $B_2 = (pIF, X, X_0, \Psi, pProt_2)$ *are faithful to* $C = (pIF, X, X_0, \Phi, \Gamma)$ *then*
$$B = (pIF, X, X_0, \Psi, pProt_1 \cup pProt_2) \text{ is also faithful to } C.$$
2. *If* $B_1 = (pIF, X, X_0, \Psi, pProt_1)$ *is faithful to* C *and* $pProt_2 \subseteq pProt_1$*, then*
$$B_2 = (pIF, X, X_0, \Psi, pProt_2) \text{ is also faithful to } C.$$
3. *If* $B_1 = (pIF, X, X_0, \Psi_1, pProt)$ *is faithful to* C *and* $\Psi_1 \sqsubseteq \Psi_2$*, then*
$$B_2 = (pIF, X, X_0, \Psi_2, pProt) \text{ is also faithful to } C.$$
4. *If a publication* B *is faithful to* C_1 *and* $C_1 \sqsubseteq C_2$*, then* B *is faithful to* C_2*.*

Proof. The first three properties are easy to check, and the last property is a direct corollary of Theorem 7.

For a contract (or a component) $K = (pIF, X, X_0, \Psi, \Gamma)$, the *largest faithful publication* of K with respect to the refinement relation is $(pIF, X, X_0, \Psi, \mathcal{PP}(K))$.

4 Primitive Open Components

The components defined in the previous section are self-contained and they implement the functionality of the services, which they provide to the clients. However, component-based software engineering is about to build new software through reuse of exiting components that are *adapted* and *connected* together. These adapters and connectors are typical *open components.* They provide methods to be called by clients on one hand, and on the other, they *require* methods of other components.

4.1 Specification of Open Components

Open components extend closed components with *required interfaces.* The body of a provided method may contain undefined methods that are to be provided when composed with other components. We therefore extend the rCOS specification notation for closed components with a declaration of a *required interface* as given in Fig. 6.

Notice that the required interface declares method signatures that do not occur in either the provided or the internal interfaces. It declares method signatures without bodies, but for generality we allow a required interface to declare its state variables too.

```
component K {
   T x = c; // state of component
   provided interface I { // provided methods
      m1(parameters) { g1 & c1 /* functionality definition */ };
      m2(parameters) { g2 & c2 /* functionality definition */ };
      ...
      m(parameters) { g & c /* functionality definition */ };
   };
   internal interface { // locally defined methods
      n1(parameters) { h1 & d1 /* functionality definition */ };
      n2(parameters) { h2 & d2 /* functionality definition */ };
      ...
      n(parameters) { h & d /* functionality definition */ };
   };
   required interface J { // required services
      T y = d;
      n1(parameters), n2(parameters), n3(parameters)
   };
   class C1{...}; class C2{...}; ... // used in the above specification
}
```

Fig. 6. Format of rCOS primitive open components

Example 3. If we "plug" the provided interface of the memory component M of Example 1 into the required interface of the following open component, we obtain an one-place buffer.

```
component Buff {
   provided interface BuffIF {
      bool r = false, w = true;
      put(Z v) { w & (W(v); r := true; w := false) }
      get(; Z v) { r & (R(; v); r := false; w := true) }
   }
   required interface BuffrIF {
      W(Z v), R(; Z v)
   }
}
```

4.2 Semantics and Refinement of Open Components

With the specification of open components using guarded commands, the denotational semantics of an open component K is defined as a functional as follows.

Definition 16 (Semantics of commands with calls to undefined methods). *Let K be a specification of an open component with provided interface $K.pIF$, state variables $K.var$, internal interface $K.iIF$ and required interface $K.rIF$, the semantics of K is the functional $[\![K]\!] : \mathcal{C}(K.rIF) \longmapsto \mathcal{C}(K.pIF)$ such* *that for each contract C in the set $\mathcal{C}(K.rIF)$ of all the possible contracts for the interface $K.rIF$, $[\![K]\!](C)$ is a contract in the set $\mathcal{C}(K.pIF)$ of all contracts for the interface $K.pIF$ defined by the specification of the closed component $K(C)$ in which*

1. *the provided interface $K(C).pIF = K.pIF$,*
2. *the state variables $K(C).var = K.var$, and*
3. *the internal interface $K(C).iIF = K.iIF \cup K.rIF$, where the bodies of the methods in $K.rIF$ are now defined to be their guarded designs given in C.*

To illustrate the semantics definition, we give the following example.

Example 4. With the memory component in Example 1, *Buff*(M) for *Buff* in Example 3 is equivalent to the contract of a one-place buffer whose publication can be specified as

```
component B {
   provided interface BuffIF {
      Z d;
      put(Z v) { d := v }
      get(; Z v) { v := d }
   }
   provided protocol {
      (?put ?get)*+(?put ?get)*?put // data parameters are omitted
   }
}
```

Definition 17. *Let K_1 and K_2 be specifications of open components with the same provided and required interfaces, respectively. K_2 is a* **refinement** *of K_1, $K_1 \sqsubseteq K_2$, if $K_1(C) \sqsubseteq K_2(C)$ holds for any contract C of the required interface of K_1 and K_2.*

The following theorem is used to establish the refinement relation of instantiated open components.

Theorem 12. *Let K be a specification of open components. For two contracts C_1 and C_2 of the required interface $K.rIF$, if $C_1 \sqsubseteq C_2$ then $K(C_1) \sqsubseteq K(C_2)$.*

To establish a refinement relation between two concretely given open components $C_1 \sqsubseteq C_2$, a refinement calculus with algebraic laws of programs are useful, e.g. $c; n(a, y) = n(a, y); c$ for any command if a and y do not occur in command c. However, the above denotational semantic semantics is in general difficult to use for checking of one component refines another or for verification of properties.

We define the notion of contracts for open components by extending the semantics of sequential programs and reactive programs to those programs in which commands contain invocations to undefined methods declared in the required interface of an open component.

Definition 18 (Design with history of method invocations). *We introduce an auxiliary state variable sqr, which has the type of sequences of method invocation symbols and define the design of a command that contains invocations to undefined methods as follows,*

- *the definition of an assignment is the same as it is defined in Sect. 2:* $x := e = \{x\} : \mathit{true} \vdash x' = e$*, implying an assignment does not change sqr,*
- *each method invocation to an undefined method* $n(T_1\ x; T_2\ y)$ *is replaced by a design*
$$\{sqr, y\} : \mathit{true} \vdash y' \in T_2 \wedge sqr' = sqr \cdot \{n(x)\},$$
where $\cdot$ *denotes concatenation of sequences, and*
- *the semantics for all sequential composition operations, i.e., sequencing, conditional choice, non-deterministic choice, and recursion, are not changed.*

A sequential design that has been enriched with the history variable sqr introduced above can then be lifted to a reactive design using the lifting function of Sect. 2.3.

With the semantics of reactive commands, we can define the semantics of a provided method $m()\{c\}$ in an open component. Also, given a state s of the component, the execution of an invocation to $m()$ from s will result in a set of sequences of possible (because of non-determinism) invocations to the required methods, recorded as the value of *sqr* in the post-state, denoted by $sqr(m(), s)$.

Definition 19 (Contract of open component). *The* **contract** $\overline{K}$ *of an open component K is defined analogously to that of a closed component except that the semantics of the bodies of provided methods are enriched with sequence observables as defined in Definition 18.*

For further understanding of this definition, let us give the weakest assumption on behavior of the methods required by an open component. To this end, we define the *weakest terminating contract*, which is a contract without side-effects, thus leaving all input variables of a method unchanged, and setting its output to an arbitrary value. The weakest terminating contract $wtc(rIF)$ of the required interface *rIF* is defined such that each method $m(x; y) \in rIF$ is instantiated with
$$m(x; y)\{\mathit{true}\ \&\ (\mathit{true} \vdash x' = x)\}.$$

Thus, $wtc(rIF)$ accepts all invocations to its methods and the execution of a method invocation always terminates. However, the data functionally is unspecified.

Proposition 1. *We have the following conjectures, but their proofs have not been established yet.*

1. *Given two open components* K_1 *and* K_2*,* $K_1 \sqsubseteq K_2$ *if* $\overline{K_1} \sqsubseteq \overline{K_2}$*.*
2. $\overline{K}$ *is equivalent to* $\overline{K(wtc(K.rIF))}$*.*

```
publication K {
   T x = c; // initial state of component
   provided interface I { // provided methods
      m1(parameters) { c1 /* unguarded design */ };
      m2(parameters) { c2 /* unguarded design */ };
      ...
      m(parameters) { c /* unguarded design */ };
   };
   internal interface {
      n1(parameters) { d1 /* unguarded design */ };
      n2(parameters) { d2 /* unguarded design */ };
      ...
      n(parameters) { d /* unguarded design */ };
   };
   required interface J { // required services
      T y = d;
      n1(parameters), n2(parameters), n3(parameters)
   };
   provided protocol {
      L1 // a regular language expression over provided method invocations
   }
   required protocol {
      L2 // a regular language expression over required method invocations
   }
   class C1{...}; class C2{...}; ... // used in the above specification
}
```

Fig. 7. Format of rCOS open publication

4.3 Transition Systems and Publications of Open Components

Given an open component K, let

- $pE(K) = \{m(u) \mid m(T_1\ x; T_2\ y) \in K.pIF \wedge u \in T_1\}$, and
- $rE(K) = \{n(u) \mid n(T_1\ x; T_2\ y) \in K.rIF \wedge u \in T_1\}$

be the possible incoming method invocations and outgoing invocations to the required methods, respectively. Further, let $\Omega(K) = pE(K) \times 2^{rE(K)^*}$. With this preparation, we can define the transition systems of open components:

Definition 20 (Transition system of open component). *Let K be an open component, we define the labeled state transition system*

$$\underline{K} = \langle K.var, K.init, \Omega(K), \emptyset \rangle,$$

such that $s \xrightarrow{m(u)/E} s'$ is a transition from state s to state s' if

- *$(s, s') \models c[u/x, v/y']$, where c is the semantic body of the method $m()$ in $\overline{K}$, and*

- *E is the set of sequences of invocations to methods in $K.rIF$, recorded in sqr in the execution from state s that leads to state s'. Here the states of $\underline{K}$ do not record the value of sqr as it is recorded in the events of the transition.*

Notice E in $s \xrightarrow{m(u)/E} s'$ is only the set of possible traces of required method invocations from s to s', not from the initial state of the transition system $\underline{K}$. The definition takes non-determinism of the provided methods into account. It shows that each state transition is triggered by an invocation of a method in the provided interface. The execution of the method may require a set of possible sequences of invocations to the methods in the required interface. Therefore, we define the following notions for open component K.

- For each trace $tr = a_1/E_1 \dots a_k/E_k$, we have a provided trace $tr^> = a_1 \dots a_k$ and sets of required traces $tr^< = E_1 \cdots E_k$, where $\cdot$ is the concatenation operation on set of sequences.
- For each provided trace pt, $\mathcal{Q}(pt) = \bigcup\{tr^< \mid tr \in \mathcal{T}(\overline{K}), tr^> = pt\}$ is the set of all corresponding required traces of pt.
- A provided trace pt is a **non-blocking provided trace** if for any trace tr such that $tr^> = pt$, tr is a non-blocking trace of $\overline{K}$.
- The provided protocol of K, denoted by $\mathcal{PP}(K)$ is the set of all non-blocking provided traces.
- The required protocol of K is a union of the sets of required traces of non-blocking provided traces $\mathcal{RP}(K) = \bigcup_{pt \in \mathcal{PP}(K)} \mathcal{Q}(pt)$.

The model of an open component is a natural extension to that of a closed component, and a closed component is a special case when the required interface is empty. Consequently, the set of required traces of a closed component is empty.

As shown in Fig. 7, the specification of a publication of an open component is similar to that of a closed component, except that the bodies of the methods of the provided and internal interfaces are defined as commands without guards, and the specification is extended with provided and required protocols.

Example 5. The publication of the open component *Buff* of Example 3 can be specified as follows.

```
publication BuffP {
   provided interface BuffIF {
      put(Z v) { W(v) }
      get(; Z v) { R(; v) }
   };
   required interface BuffrIF { W(Z v), R(; Z v) };
   provided protocol { (?put ?get)*+(?put ?get)*?put };
   required protocol { (!W !R)*+(!W !R)*!W }
}
```

Note, unlike in a contract of a component where each transition step is represented atomically, in a publication an action a/E is executed as an atomic step of state transition, though E represents a set of traces of method invocations.

However, the composability can be checked in the following way: If the provided protocol $K.pProt$ of a component K contains (accepts) all the invocation traces of the required protocol $\mathcal{RP}(J)$ of an open component J, then K can be plugged to provide the services that J requires.

5 Processes

All components that we have defined so far are passive in the sense that a component starts to execute only when a provided method is invoked from the environment (say, by a client). Once a provided method is invoked, the component starts to execute the body of the method, provided it is enabled. The execution of the method is atomic and follows the *run to complete semantics.* However, it is often the case that *active software entities* are used to coordinate the components when the components are being executed. For example, assume we have two copies of component *Buff* in Example 3, say B_1 and B_2 whose provided interfaces are the same as *Buff*, except for *put* and *get* being renamed to put_i and get_i for B_i, respectively, where $i = 1, 2$. We can then write a program P that repeatedly calls $get_1(;a); put_2(a)$ when both get_1 and put_2 are enabled. Then, P *glues* B_1 and B_2 to form a two-place buffer. We call such an active software entity a *process.*

5.1 Specification of Processes

In this section, we define a class of processes that do not provide services to clients but only actively calls provided services of other components. In the rCOS specification notation, such a process is specified in the format shown in Fig. 8. In the body of an action (which does not contain parameters), there are calls to methods in both the internal interface section and the required interface section, but not to other methods.

5.2 Contracts of Processes

Notice that the actions, denoted by $P.ifa$, are autonomous in the sense that when being enabled they can be non-deterministically selected to execute. The execution of an action is atomic and may involve invocations to methods in the required interface $P.rIF$, as well as program statements and invocations to methods defined in the internal interface $P.iIF$. We will see later when we define the composition of a component and a process that execution of an atomic action a in P *synchronizes* all the executions of required methods contained in a, i.e., the execution of a locks all these methods until a terminates. For instance, in the two place buffer example at the beginning of this section, $get_1(;a); put_2(a)$ is the only action of the process P. When this action is being executed, B_1 cannot execute another *get* until this action finishes.

The denotational semantics of a process P is similar to that of an open component in the sense that it is a functional over the set $\mathcal{C}(P.rIF)$ of the contracts

```
process P {
   T x = c; // initial state of process
   actions { // guarded commands
      a1 { g1 & c1 };
      ...
      ak { gk & ck }
   };
   required interface J { // required services
      T y = d;
      n1(parameters), n2(parameters), n3(parameters)
   };
   internal interface { // locally defined methods
      n1(parameters) { h1 & d1 /* functionality definition */ };
      n2(parameters) { h2 & d2 /* functionality definition */ };
      ...
      n(parameters) { h & d /* functionality definition */ };
   };
   class C1{...}; class C2{...}; ... // used in the above specification
}
```

Fig. 8. Format of rCOS process specifications

of interface $P.rIF$ such that for each contract C in $\mathcal{C}(P.rIF)$, $[\![P]\!](C)$ is a fully defined process, called a *self-contained process*, containing the autonomous actions $P.ifa$. In this way, a failure-divergence semantics in terms of actions in $P.ifa$ and a refinement relation can be defined following the definitions of Sect. 2.

However, we apply the same trick as we did when defining the semantics in Definition 18 for the body of a provided method in an open component, which contains calls to undefined methods. Therefore, the execution of an atomic action a in a process from a state s records the set sqr of possible sequences of invocations to methods declared in the required interface.

Definition 21 (Contract of process). *Given a specification of a process P in the form shown in Fig. 8, its* **contract** $\overline{P}$ *is defined analogously to Definition 19 by enrichment with history variables, i.e., it is specified as shown in Fig. 9.*

Example 6. Consider two instances of the *Buff* component, B_1 and B_2, obtained from *Buff* by respectively renaming *put* to put_1 and put_2 as well as *get* to get_1 and get_2. We design a process that keeps getting an item from B_1 and putting it into B_2 when get_1 and put_2 are enabled. The contract of the process is specified as follows.

```
process P {
   T x = c; // initial state of process
   actions { // reactive designs
      a1 { /* $g_1$ & $c_1$ design enriched by history variables $sqr$ */ };
      ...
      ak { /* $g_k$ & $c_k$ design enriched by history variables $sqr$ */ }
   };
   required interface J { // required services
      T y = d;
      n1(parameters), n2(parameters), n3(parameters)
   };
   class C1{...}; class C2{...}; ... // used in the above specification
}
```

Fig. 9. Format of rCOS process contracts

```
process Shift {
   T x = c; // state of process
   actions { // reactive designs
      move { {sqr}: (get1(; x); put2(x) };
           // equals to $true \vdash sqr' = \{get_1(;a) \cdot put_2(a) \mid a \in Z\}$
   }
   required interface J { // required services
      get1(; Z x), put2(Z x)
   };
}
```

Notice that there is no guard for the process in the above example, it will be enabled whenever its environment are ready to synchronize on the required methods, i.e., they are enabled in their own flows of execution. Now we are ready to define the transition system for a process, and from that, the publication of a process.

5.3 Transition Systems and Publications of Processes

Given a process P, we define the set $\omega P = 2^{P.rIF^*}$ to be the set of all sets of invocations sequences to methods in the required interface of P. Following the way in which we defined the transition system of an open component, we define the transition system of a process.

Definition 22 (Transition system of processes). *The transition system $\underline{P}$ of a process P is the quadruple $\langle P.var, P.init, \omega P, \emptyset\rangle$, where for $E \in \omega P$, states s, s' of P, and an action a of P with body c,*

$$s \xrightarrow{a/E} s' \text{ if } (s \oplus \{sqr \mapsto \emptyset\}, s' \oplus \{sqr \mapsto E\}) \vdash c$$

holds.

We can define the execution failure-divergence semantics $(\mathcal{ED}(P), \mathcal{EF}(P))$ and interaction failure-divergence semantics $(\mathcal{ID}(P), \mathcal{IF}(P))$ for process P in terms of the transition system $\underline{P}$. The interaction traces and the *failure-divergence refinement* of processes follow straightforward. However, a process can non-deterministically invoke methods of components, and its whole trace set is taken as the **required protocol**

$$\mathcal{RP}(P) = \bigcup \{E_1 \cdots E_k \mid /E_1 \cdots /E_k \in \mathcal{T}(P)\}.$$

The definition of the protocol of a process allows us to define publications of processes in the same way as we defined publications for open components.

6 Architectural Compositions and General Components

We have defined the models of primitive closed and open components as well as processes. These models do not show any entities resulting from compositions, but software components and processes that are about constructing components by compositions of these primitive forms. In this section, we go beyond the primitive component model and define architectural composition operations that allow us to build hierarchical software architectures. We will also define a general model of components in this way and discuss the special classes of components that are useful in building software components. For this, we reiterate the notations for the different sections in a component specification (or process) K, that are $K.var$, $K.init$, $K.pIF$, $K.rIF$, $K.iIF$ and $K.Act$. Component operations are syntactically defined as operations on these sections, and then their semantics definitions are derived. We provide examples to illustrate their meanings and uses, too.

6.1 Coordination of Components by Processes

One way of building larger components from existing components is to coordinate their behavior by active processes. We start with this composition as it introduces internal autonomous actions to components, by which the primitive component model defined in the previous section is extended.

Definition 23 (Coordination of components). *Let K be a component and P a process. The coordination of K by P, denoted by $K \parallel P$, is defined if K and P do not have common variables. When $K \parallel P$ is defined, it is the following* **general component***, denoted by J,*

$$\begin{aligned} &J.var = K.var \cup P.var, && J.init = K.init \times P.init, \\ &J.pIF = K.pIF, && J.rIF = K.rIF \cup P.rIF, \\ &J.iIF = K.iIF \cup P.iIF. \end{aligned}$$

Additionally, we extend the specification of components with the section

$$J.inva = P.Act$$

containing the set of actions that have the triggering events invisible to the environment and can be executed autonomously when enabled. A subset of $P.Act$ contains those which have no external required events or external triggering events. These are the actions entirely internal in the component.

It is necessary to introduce the internal autonomous actions in the above definition, because internal actions emerge when a process is used to coordinate the behavior of a component.

Definition 24 (Transition system of coordination). *When $J = K \parallel P$ is defined, we define the general transition system $\underline{J} = \langle J.var, J.init, \Omega, \Lambda \rangle$ such that*

1. $\Omega = K.ifa$, *that is* $\omega(K.pIF) \times 2^{\omega(K.rIF)^*}$,
2. $\Lambda = P.Act$,
3. $(s_1, s_2) \xrightarrow{e/E} (s_1', s_2')$ *is a transition in* $\underline{J}$ *if*

   (component step) $s_2 = s_2'$, $e \in \omega(K.pIF)$ *and* $s_1 \xrightarrow{e/E} s_1'$ *is a transition of* $\underline{K}$, *or*

   (process step) $e \in \Lambda$ *and there exists* $s_2 \xrightarrow{e/F} s_2'$ *in* $\underline{P}$ *such that for every*

$$tr_0 \cdot m_1 \cdot tr_1 \cdots m_k \cdot tr_k \in F,$$

   where $m_i \in \omega(K.pIF)$ *and* $tr_j \in \omega(J.rIF)^*$, *it holds*
   - *there exist* $E_1, \ldots, E_k$ *such that* $s_1 \xRightarrow{m_1/E_1,\ldots,m_k/E_k} s_1'$,
   - $tr_0 \cdot E_1 \cdots E_k \cdot tr_k \subseteq E$, *and*
   - E *is the smallest set that satisfies these two properties.*

In rCOS, we specify a general component in the format shown in Fig. 10, that extends the specification of an open component in Fig. 6 with a section of invisible actions. Thus, K is a primitive open component if $K.inva$ is empty, a closed component when $K.rIF$ is empty, and a process when $K.pIF$ is empty. General components thus contain both active and passive behavior. From now on, a process is also treated as a component.

As shown in the definition of $K \parallel P$, actions defined in $J.inva$ may also require methods given in the required interface $J.rIF$. When an invisible action does not require any methods outside the component, it is then totally invisible.

6.2 Composition of Processes

Now we define the parallel composition of processes. Since processes only actively call external methods, but do not provide methods to be called, two processes do not communicate directly. Thus, the execution of the parallel composition of two processes is simply the interleaving execution of the actions of the individual processes.

```
component K {
   T x = c; // initial state of component
   provided interface I { // provided methods
      m1(parameters) { g1 & c1 /* functionality definition */ };
      m2(parameters) { g2 & c2 /* functionality definition */ };
      ...
      m(parameters) { g & c /* functionality definition */ };
   };
   internal interface { // locally defined methods
      n1(parameters) { h1 & d1 /* functionality definition */ };
      n2(parameters) { h2 & d2 /* functionality definition */ };
      ...
      n(parameters) { h & d /* functionality definition */ };
   };
   actions { // invisible autonomous action
      a1() { f1 & e1 }; // no parameters
      a2() { f2 & e2 }; // no parameters
      ...
      a() { f & e }
   };
   required interface J { // required services
      T y = d;
      n1(parameters), n2(parameters), n3(parameters)
   };
   class C1{...}; class C2{...}; ... // used in the above specification
}
```

Fig. 10. Format of rCOS general components

Definition 25 (Parallel composition of processes). *For two processes P_1 and P_2, the parallel composition $P_1 \parallel P_2$ is defined if they have neither common variables nor common action names. When $P_1 \parallel P_2$ is defined, the composition, denoted by P, is defined as follows,*

$$P.var = P_1.var \cup P_2.var, \quad P.init = P_1.init \times P_2.init,$$
$$P.Act = P_1.Act \cup P_2.Act, \quad P.rIF = P_1.rIF \cup P_2.rIF.$$

The following theorem ensures that the above syntactic definition is consistent with the semantic definition.

Theorem 13 (Semantics of process parallel composition). *If $P = P_1 \parallel P_2$ is defined, the transition system $\underline{P}$ of the composition P is the product of $\underline{P_1}$ and $\underline{P_2}$, that is,*

1. *the states $\Sigma_{P.var} = \Sigma_{P_1.var} \times \Sigma_{P_2.var}$,*
2. *the initial states $P.init = P_1.init \times P_2.init$,*
3. *the transition labels $\Omega = 2^{\omega(P_1.rIF)^*} \cup 2^{\omega(P_2.rIF)^*}$, and*
4. *$(s_1, s_2) \xrightarrow{e/E} (s_1', s_2')$ is a transition of $\underline{P}$ if either*

– $s_2 = s_2'$ *and* $s_1 \xrightarrow{e/E} s_1'$ *is a transition of* $\underline{P_1}$, *or*
– $s_1 = s_1'$ *and* $s_2 \xrightarrow{e/E} s_2'$ *is a transition of* $\underline{P_2}$.

It can be shown that a parallel composition of processes preserves the refinement relation between processes.

6.3 Parallel Composition of Components

We continue with introducing composition operators to build larger components.

Definition 26 (Parallel composition of components). *Given two components K_1 and K_2, either closed or open, the parallel composition $K_1 \parallel K_2$ is defined, provided the following conditions hold,*

1. *they do not have common variables, $K_1.var \cap K_2.var = \emptyset$,*
2. *they do not have common provided methods, $K_1.pIF \cap K_2.pIF = \emptyset$,*
3. *they do not have common autonomous actions, $K_1.Act \cap K_2.Act = \emptyset$, and*
4. *they do not have common internal methods, $K_1.iIF \cap K_2.iIF = \emptyset$.*

When the composition is defined, the composed component, denoted by K, is defined as

$$\begin{aligned}
&K.var = K_1.var \cup K_2.var, && K.init = K_1.init \times K_2.init,\\
&K.iIF = K_1.iIF \cup K_2.iIF, && K.pIF = K_1.pIF \cup K_2.pIF,\\
&K.rIF = (K_1.rIF \cup K_2.rIF) \setminus (K_1.pIF \cup K_2.pIF), &&\\
&K.Act = K_1.Act \cup K_2.Act. &&
\end{aligned}$$

It is important to note that this syntactic definition is actually consistent with the semantic definition by the transition systems.

Definition 27 (Parallel composition of component transition systems). *The labeled transition system $\underline{K} = \langle K.var, K.init, \Omega, K.Act\rangle$ of the parallel composition $K = K_1 \parallel K_2$ is defined by*

1. *Ω is defined from the interface K as for a primitive component,*
2. *$(s_1, s_2) \xrightarrow{a/E^-} (s_1', s_2')$ is a transition of $\underline{K}$ if one of the following conditions holds.*
   (K_1 step) When $a \in \omega(K_1.pIF)$
   (a) there exists a transition $s_1 \xrightarrow{a/E} s_1'$ of $\underline{K_1}$, with
   $$tr_0 \cdot m_1 \cdot tr_1 \cdots m_k \cdot tr_k \in E,$$
   where $m_i \in \omega(K_2.pIF)$ and $tr_i \in \omega(K.rIF)^*$ for $0 \le i \le k$, then
   (b) if $k = 0$, $s_2' = s_2$
   (c) for each $s_2 \xRightarrow{m_1/E_1,\ldots,m_k/E_k} s$ in K_2 (i.e., $k > 0$)
   – *$s_2' =$ **error** if $(E_1 \cup \cdots \cup E_k) \cap \omega(K_1.pIF) \neq \emptyset$, and*

– *$s'_2 = s$ otherwise,*

(d) $tr_0 \cdot E_1 \cdot tr_1 \cdots E_k \cdot tr_k \subseteq E^-$, and

(e) E^- is the smallest set that satisfies above conditions.

(K_2 step) When $a \in \omega(K_2.pIF)$, the transition is defined symmetrically.

(action step) When $a \in K.Act$, the transition is defined like the process step in Definition 24 for the coordination of a component by a process. Here, the autonomous actions K can be seen as a process step and the rest can be seen as a component step.

The **error** state is a designated deadlock state used to explicitly mark failures from cyclic method calls in compositions violating the *run to complete semantics*. We define a composed state (s_1, s_2) an **error** state if either s_1 or s_2 is **error**. We will discuss the nature of the **error** state in more detail in the next section.

Notice that the required interfaces of K_1 and K_2 do not have to be disjoint, they may require common services. A special case for the parallel composition is that when K_1 and K_2 are totally disjoint, i.e., there is no overlapping in the required interfaces and no component provides methods that the other requires. In this case, we call $K_1 \parallel K_2$ a *disjoint union*, and denote it by $K_1 \otimes K_2$. Even more specifically, when K_1 and K_2 are closed components, $K_1 \parallel K_2$ is always a disjoint union and $K_1 \otimes K_2$ is also a closed component.

For the refinement of components, we have that $\parallel$ is monotonic.

Theorem 14 (Parallel composition preserves refinement). *If $K_1 \sqsubseteq J_1$ and $K_2 \sqsubseteq J_2$, then $K_1 \parallel K_2 \sqsubseteq J_1 \parallel J_2$.*

The parallel composition of components is commutative. Since methods from the provided interface are never hidden from a composition, it is also associative.

Theorem 15. *$K_1 \parallel K_2 = K_2 \parallel K_1$ and $J \parallel (K_1 \parallel K_2) = (J \parallel K_1) \parallel K_2$.*

6.4 Renaming and Restriction

When we compose components (including processes), sometimes the provided method names and required method names do not match. We often need to rename some methods. A rename function for a component is a one-one function on the set of methods.

Definition 28 (Renaming). *Let K be a component and f a renaming function, we use $K[f]$ to denote the component for which all the specifications are the same as those of K, except for the provided and required interfaces, which are defined by*

1. *$K[f].pIF = \{f(m) \mid m \in K.pIF\}$,*
2. *$K[f].rIF = \{f(m) \mid m \in K.rIF\}$, and*
3. *any occurrence of m in K is replaced by $f(m)$.*

Notice that we do not allow renaming internal interface methods, thus an implicit assumption is that a provided or required interface method is not renamed to an internal interface method. This is equivalent to require that $f(m) = m$ for all $m \in K.iIF$.

As a component only involves a finite number of methods, thus a renaming function is only effective on these methods. Therefore, for any renaming functions f and g, if $f(m) = g(m)$ for any method name m of K, then $K[f] = K[g]$. In particular, $K[n/m]$ is the component obtained from K by renaming its method m to n. This is extended to the case when a number of methods are renamed, i.e., $K[n_1/m_1, \ldots, n_k/m_k]$, which is similar to the renaming function in process algebras.

Example 7. For the memory component M in Example 1, $M[put/W, get/R]$ is the same as M except that any occurrence of the method name W is replaced by *put* and any occurrence of the method name R is replaced by *get*.

It is often the case when using a component in a context that some provided methods are restricted from being accessed by the environment. However, these methods cannot be simply removed. Instead, they should be moved to the internal interface so that they can be still called by the other provided and internal methods of the same component.

Definition 29 (Restriction). *Let K be a component and β a subset of the names of the provided methods of K, the component $K \setminus \beta$ is obtained from K by moving the declarations of the methods in β from the provided interface section to the internal interface section, that is,*

$$(K \setminus \beta).pIF = K.pIF \setminus \{m(u; v)\{c\} \mid m \in \beta \wedge m(u; v)\{c\} \in K.pIF\},$$
$$(K \setminus \beta).iIF = K.iIF \cup \{m(u; v)\{c\} \mid m \in \beta \wedge m(u; v)\{c\} \in K.pIF\}.$$

Example 8. Let M be the memory component given in Example 1, *Buff* the open component in Example 3, and $B = (M \parallel \mathit{Buff}) \setminus \{W, R\}$. Thus, B is a one-place buffer component. Further, let $B_i = B[get_i/get, put_i/put]$, for $i = 1, 2$. We now use process *Shift* to coordinate $B_1 \otimes B_2$ and define

$$\mathit{Buff}_1 = ((B_1 \otimes B_2) \parallel \mathit{Shift}) \setminus \{get_1, put_2\},$$

Buff_1 is a two-place buffer.

Notice that in the above example, the closed component M provides all the methods required by the open component *Buff*. The restriction of $\{W, R\}$ from the composition $M \parallel \mathit{Buff}$ makes W and R only accessible to *Buff*. We call such a restricted composition *plugging*, and denote it by $M \gg \mathit{Buff}$. In general, we have the following definition.

Definition 30 (Plugging). *Let K_1 and K_2 be components such that $K_1 \parallel K_2$ is defined. If $K_2.rIF \subseteq K_1.pIF$, then we define the* plugging of K_1 with K_2 *by*

$$K_1 \gg K_2 = (K_1 \parallel K_2) \setminus K_2.rIF.$$

In the following example, we build the two-place buffer in a different way.

Example 9. We first define the following open component.

```
component Connector {
   int z;
   provided interface { shift() { get1(; z); put2(z) } };
   required interface { get1(; int z); put2(int z) }
}
process P {
   required interface { shift() }
}
```

Then, we can define the component

$$\mathit{Buff}_2 = ((B_1 \otimes B_2) \gg \mathit{Connector} \parallel P) \setminus \{\mathit{shift}\}.$$

This can be simply written as $\mathit{Buff}_2 = ((B_1 \otimes B_2) \gg \mathit{Connector}) \gg P$. One can prove that Buff_2 is equivalent to the component Buff_1 in Example 8.

When a number of components are coordinated by a process, the components are not aware of which other components they are working with or exchange data with. Another important aspect is the separation of local data functionality of a component from the control of its interaction protocols. We can design components in which the provided methods are not guarded and thus have no access control. Then, using connectors and coordinating processes the desired interface protocols can be designed. In terms of practicability, most connectors and coordinating processes in applications are data-less, thus having a purely event-based interface behavior. This allows rCOS to enable the separation of design concerns of data functionality from interaction protocols.

6.5 More Examples

The memory component M given in Example 1 models a perfect memory cell in the sense that its content will not be corrupted. As in Liu's and Joseph's work on fault-tolerance [44], a fault can be modeled as an internal autonomous action. We model a faulty memory, where an occurrence of a fault corrupts the content of the memory.

```
component fM { // faulty memory
   provided interface MIF {
      Z d; bool start = false;
      W(Z v) { true & (d := v; start := true) }
      R(; Z v) { start & v := d }
   };
   actions {
      fault() { true & true ⊢ d' ≠ d } // corrupting the memory
   }
}
```

Now we show how to use three faulty memories to implement a perfect memory. First, for $i = 1, 2, 3$, let $fM_i = fM[W_i/W, R_i/R]$. We define the following open component.

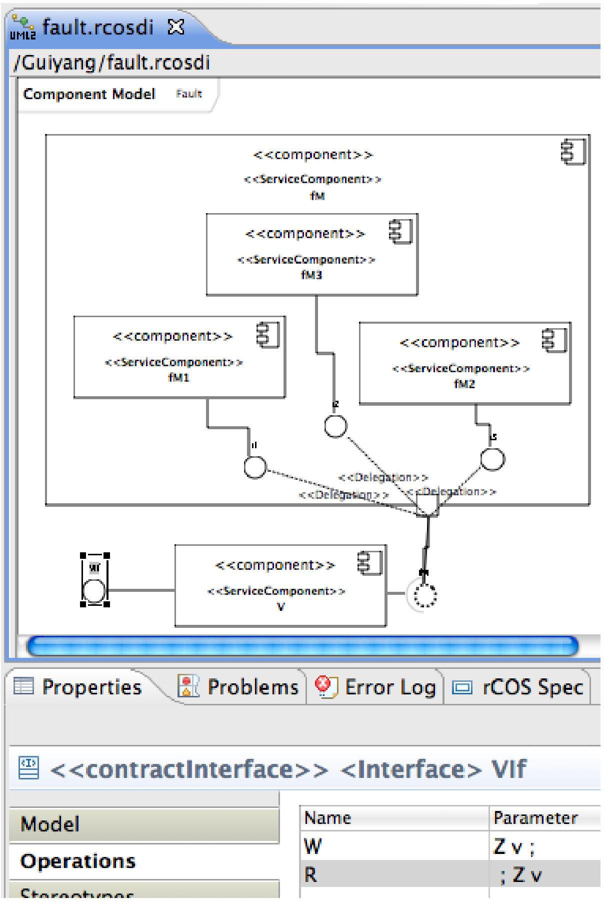
```
component V { // majority voting
   provided interface VIF {
      W(Z v) { W1(v); W2(v); W3(v) }
      R(; Z v) {
         var Z v1, Z v2, Z v3;
         R1(; v1); R2(; v2); R3(; v3);
         vote(v1, v2, v3; v);
         end v1, v2, v3
      }
   }
   required protocol { // interleaving of all fM_i's provided protocols
      ...
   }
}
```

Then we have the composite component $(fM_1 \parallel fM_2 \parallel fM_3) \gg V$, that can be proven to be equivalent to the memory component M in Example 1. The proof requires an assumption that at any time at most one memory is faulty. This involves the use of auxiliary variables to record the occurrence of the fault [44]. The architecture of this fault-tolerant memory is shown in the component diagram in Fig. 11, which is a screen-shot from the rCOS Modeler.

In this section, we have defined the general parallel composition for components (including closed components, open components and processes). However, it is important to develop a theory of compositions, techniques for checking composabilities among components, and refinement calculi for the different models with respect to parallel composition and restriction. These have been partly studied in the rCOS literature [12, 19, 20, 25, 26, 45, 47, 62, 66]. However, the semantic models defined in this chapter extend the models in those papers. Thus, the theory and techniques of refinement and composability need a reinvestigation. In the following section, we present preliminary work on how an interface model of components supports composability checking, focusing on the interaction between components.

7 Interface Model of Components

In rCOS, the refinement of data functionality is dealt within the unified semantic theory for sequential programming presented in Sect. 2. Interactions are handled with the failure-divergence semantics of components. In this section, we first present a model of components that abstracts the data states away, thus focusing only on interactions. We call this model *component automata*. This model still exhibits non-determinism caused by autonomous actions and encounters difficulties in checking composability, for third party composition in particular. Therefore, we will define an interface model for components, called *interface*



<<contractInterface>> <Interface> VIf

Name	Parameter
W	Z v ;
R	; Z v

Fig. 11. Component-based design of a fault-tolerant component

publication automata. An interface publication automaton is *input-deterministic*, that is, at each step the choice among provided method invocations is controlled by the environment. Both models are simplified labeled transition systems, but the states are only symbolic states for control of dynamic flows. We also focus on finite state components.

7.1 Component Automata

First, some preliminary notations are defined that we are going to use in the discussion in this section. For a pair $\ell = (e_1, e_2)$ in a product of sets $A_1 \times A_2$, we define the projection functions π_i, for $i = 1, 2$, that is, $\pi_1(\ell) = e_1$ and $\pi_2(\ell) = e_2$. The projection functions are naturally extended to sequences of pairs and sets of sequences of pairs as follows. Given a sequence of pairs $tr = \langle \ell_1, \ldots, \ell_k \rangle$

and a set T of sequences of pairs, we define $\pi_i(tr) = \langle \pi_i(\ell_1), \ldots, \pi_i(\ell_k) \rangle$, and $\pi_i(T) = \{\pi_i(tr) \mid tr \in T\}$, for $i = 1, 2$.

Given $\Gamma \subseteq \Omega$ and a sequence $tr \in \Omega^*$, $tr\lfloor_\Lambda$ is the restriction of tr to element in Λ, returning the sequence obtained from tr by keeping only those elements in Λ. We also extend the restriction function to sets of sequences and define $T\lfloor_\Gamma = \{tr\lfloor_\Gamma \mid tr \in T\}$.

We now introduce a symbolic version of the labeled transition systems, called component automata, by replacing the variables with a set of states and abstract the data parameters from the interface methods. We only consider finite state automata.

Definition 31 (Component automaton). *A* **component automaton** *is a tuple $K = \langle \Sigma, s_0, pIF, rIF, Act, \delta \rangle$, where*

- *Σ is a finite set of states and $s_0 \in \Sigma$ is the initial state;*
- *pIF, rIF, and Act are disjoint finite sets of* **provided**, **required** *and* **internal** *events, respectively;*
- *$\delta \subseteq \Sigma \times \Omega(pIF, rIF, Act) \times \Sigma$ is the transition relation, where the set of* **labels** *is defined as $\Omega(pIF, pIF, Act) = (pIF \cup Act) \times (2^{rIF^*} \setminus \{\emptyset\})$, simply denoted by Ω when there is no confusion.*

As before, we use e/E to denote a pair (e, E) in Ω and a transition $(s, e/E, s') \in \delta$ by $s \xrightarrow{e/E} s'$. This transition is called a step of *provided transition* if $e \in pIF$, otherwise an *autonomous transition*. We use $s \xrightarrow{e/} s'$ for $s \xrightarrow{e/\{\epsilon\}} s'$. Notice that $e/\emptyset$ is not a label in an automaton, and a transition without required events is a transition by a label of the form $e/\{\epsilon\}$. The internal events are prefixed with a semicolon, e.g., $s \xrightarrow{;e/E} s'$, to differentiate them from a transition step by a provided event. We use τ to represent an internal event when it causes no confusion. For a state s, we define the set of events with outgoing transitions by

$$out(s) = \{e \in pIF \cup Act \mid \exists s', E \bullet s \xrightarrow{e/E} s'\}.$$

Further, let $out^\circ(s) = out(s) \cap Act$ and $out^\bullet(s) = out(s) \cap pIF$. We write $s \xrightarrow{e/} s'$ for $s \xrightarrow{e/E} s'$, when E is not significant.

Notice that there are no guards for transitions. Instead, the guards of actions from the general component transition systems are encoded in the states of the automata. A composite event e/E in Ω is *enabled* in a state s if there exists a transition $s \xrightarrow{e/E} s'$, and an action $e \in pIF \cup Act$ is enabled in a state $s \in \Sigma$, if there is a set of sequences $E \in 2^{rIF^*}$ such that $s \xrightarrow{e/E} s' \in \delta$. Then, the executions of an automaton C can be defined in the same way as for a labeled transition system. However, the execution traces and the interaction traces are of no significant difference. Formally, we have the following definitions and notations,

- a sequence of transitions $s \xrightarrow{\ell_1} s_1 \cdots \xrightarrow{\ell_k} s'$ is called an execution sequence, and $\langle \ell_1, \ldots, \ell_k \rangle$ is called a trace from s to s',

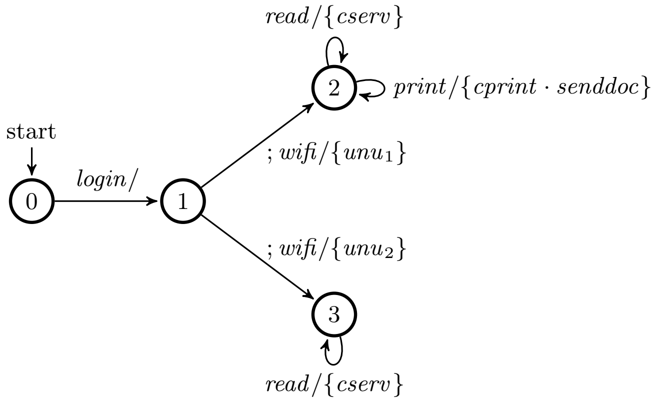


Fig. 12. Automaton of internet connection component C_{ic}

- we write $s \overset{\ell_1,\ldots,\ell_k}{\Longrightarrow} s'$ if there exists an execution sequence $s \xrightarrow{\ell_1} s_1 \cdots \xrightarrow{\ell_k} s'$,
- for a trace tr over Ω and a state s, $target(tr, s) = \{s' \mid s \overset{tr}{\Longrightarrow} s'\}$, and $target(tr) = target(tr, s_0)$,
- for a sequence sq over $pIF \cup Act$, we write $s \overset{sq}{\Longrightarrow} s'$ if there is a trace tr such that $s \overset{tr}{\Longrightarrow} s'$ and $\pi_1(tr) = sq$,
- $\mathcal{T}(s) = \{\langle \ell_1, \ldots, \ell_k \rangle \mid \exists s' \bullet s \overset{\ell_1,\ldots,\ell_k}{\Longrightarrow} s'\}$, and it is called the traces of s,
- $\mathcal{T}(s_0)$ is the set of traces of the component automaton C, it is also denoted by $\mathcal{T}(C)$,
- for a state s, the *provided traces* for s are given by

$$\mathcal{PT}(s) = \{\pi_1(tr){\restriction}pIF \mid tr \in \mathcal{T}(s)\},$$

- the set $\mathcal{PT}(s_0)$ is called the set of *provided traces* of C, and it is also written as $\mathcal{PT}(C)$.

Example 10. Consider the internet-connection component presented in Fig. 12. It provides the services *login*, *print*, and *read* to the environment and there is an internal service ; *wifi*. The services model the login into the system, invocation of printing a document, an email service, and automatically connecting the wifi, respectively. The component calls the services unu_1, unu_2, *cserv*, *cprint* and *senddoc*. The first three of them model the searching for a wifi router nearby, connecting to either the unu_1 or unu_2 wireless network, and then to an application server, respectively. The services *cprint* and *senddoc* connect to the printer, send the document to print and start the printing job. The *print* service is only available for the wifi network unu_1, while *read* can be accessed from both networks.

The component automaton in Fig. 12 can perform, for example, the following execution,

$$\langle 0, (login/\{\epsilon\}), 1, (;wifi/\{unu_1\}), 2, (print/\{cprint \cdot senddoc\}), 2\rangle.$$

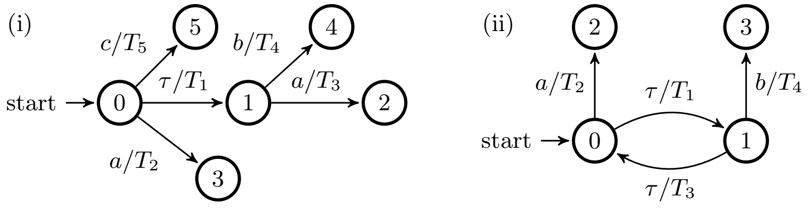


Fig. 13. Examples of enabled actions

Now $pt = \langle login, print\rangle$ is a provided trace of the execution and the set of required traces of pt is $\{\langle unu_1 \cdot cprint \cdot senddoc\rangle\}$.

7.2 Non-blockable Provided Events and Traces

The model of component automata describes how a component interacts with the environment by providing and requiring services. However, some provided transitions or executions may be blocked due to the non-determinism caused by autonomous actions. In this section, we will discuss about the non-refusal of provided events and traces.

Figure 13 shows what kinds of provided events can be refused (or blocked). The states 0 and 1 in either automaton are indistinguishable, because every internal autonomous transition (a τ transition) will be taken eventually. Therefore, a state, at which an internal autonomous transition may happen, like state 0, is called an *unstable state*. In general, a state s of a component automaton C is *stable* if $out^\circ(s) = \emptyset$, that is, no internal actions are enabled, otherwise s is unstable. A state s is a *deadlock state* if there is no action enabled at all. A deadlock state is always a stable state.

Now consider state 1 in the automaton in Fig. 13(i), it is a stable state. The automaton will eventually leave the initial state 0, because if a or c are never tried, the autonomous τ transition will be eventually performed. Thus the provided action b cannot be refused (blocked) in state 0 or 1, because it is enabled in state 1. We call a state s' *internally reachable* from state s, denoted by $autoR(s, s')$, if there is a sequence (possibly empty and in that case $s' = s$) $s \xRightarrow{\tau_1,\ldots,\tau_k} s'$ of internal transitions from s to s'. We can see that in the automaton in Fig. 13(i) state 1 is internally reachable from state 0 and all internal executions (only one in this case) reach state 1. For this reason, we say provided action b cannot be refused from state 0. Notice that in the automaton in Fig. 13(ii), there is no internally reachable stable state from 0.

We define $autoR(s) = \{s' \mid autoR(s, s')\}$ to be the set of internally reachable states from s, and $autoR^\bullet(s)$ the set of internally reachable stable states, i.e., $\{s' \mid s' \in autoR(s) \textit{ and } s' \text{ is stable}\}$. Notice that $autoR(s, s')$ is a transitive relation and $autoR(s)$ is closed under this relation. Thus, $autoR^\bullet(s) = \emptyset$ if there exists a livelock state in $autoR(s)$, i.e., an infinite sequence of internal transitions is possible.

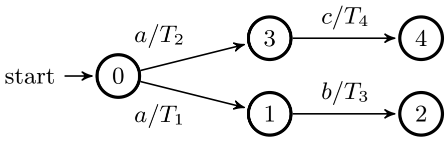


Fig. 14. An example of a refusal trace

With the above discussion, we are ready to define the refusal provided events of a component automaton. Informally, a provided event in *pIF* is a *refusal* of state s if it is not enabled at one of the internally reachable stable states s' of s.

Definition 32 (Refusal events). *Let s be a state of a component automaton C, the set of* **local refusal events** *at s is*

$$\mathcal{R}(s) = \{e \mid e \in pIF \wedge \exists s' \in autoR^{\bullet}(s) \bullet e \text{ is disabled at } s'\}.$$

We use $\overline{\mathcal{R}}(s)$ to denote the set of **local non-refusal** *(non-blockable) events at s.*

The important concept we are developing in this subsection is the notion of *non-refusal traces*, that is going to be used for the publication of a component.

Consider the component automaton shown in Fig. 14. The provided event a is enabled at state 0, however, after the invocation of a, the component determines internally whether to move to state 1 or 3. So, both of b and c may be refused after a.

Definition 33 (Non-blockable provided trace). *Let $\langle a_1, \cdots, a_k\rangle$, $k \geq 0$, be a sequence of provided events of a component automaton C. It is called a* **non-blockable provided trace** *at state s if for $0 \leq i \leq k-1$ and any state s' such that $s \overset{tr}{\Longrightarrow} s'$ and $\pi_1(tr)\lfloor pIF = \langle a_1, \ldots, a_i\rangle$, a_{i+1} is not a refusal at s', i.e., $a_{i+1} \in \overline{\mathcal{R}}(s')$.*

A trace tr of a component automaton C is *non-blockable* at a state s, if the provided trace $\pi_1(tr)\lfloor_{pIF}$ is non-blockable at s. We use $\mathcal{PP}(s)$ and $\mathcal{UT}(s)$ to denote the set of all non-blockable provided traces (also denoted as provided protocols like in the previous sections) and non-blockable traces at state s, respectively. When s is the initial state of C, we also write $\mathcal{PP}(s)$ and $\mathcal{UT}(s)$ as $\mathcal{PP}(C)$ and $\mathcal{UT}(C)$, respectively.

7.3 Interface Publication Automata

We now define a model of input-deterministic component automata that have non-blockable traces only. We use this model for *publications of components* as they give better composability checking. The main result of this subsection is the design of an algorithm that transforms a general component automaton to such an *interface publication automaton.*

Definition 34 (Input-determinism). *A component automaton*

$$C = \langle S, s_0, P, R, A, \delta \rangle$$

is input-deterministic *if for any* $s_0 \overset{tr_1}{\Longrightarrow} s_1$ *and* $s_0 \overset{tr_2}{\Longrightarrow} s_2$ *such that*

$$\pi_1(tr_1)\lfloor_{pIF} = \pi_1(tr_2)\lfloor_{pIF},$$

the sets of non-blockable events are identical, i.e., $\overline{R}(s_1) = \overline{R}(s_2)$.

The definition says that any provided event e is either a refusal or a non-refusal at both of any two states s_1 and s_2 that are reachable from the initial state through the same provided trace. Therefore, any provided trace of an input-deterministic component automaton is not blocked, provided the required events are acceptable by the environment. Thus, we call an input-deterministic automaton an *interface publication automaton.*

The following theorem states that all the traces of an input-deterministic component automaton are non-blockable.

Theorem 16. *A component automaton* C *is input-deterministic iff* $\mathcal{PT}(C) = \mathcal{PP}(C)$.

Proof. $\mathcal{PT}(C) = \mathcal{PP}(C)$ means every provided trace of C is non-blockable actually.

First, we prove the direction from left to right. From the input-determinism of C follows that for each provided trace $pt = (a_0, \ldots, a_k)$ and each state s with $s_0 \overset{tr}{\Longrightarrow} s$ and $\pi_1(tr) = \langle a_0, \ldots, a_i \rangle$ for $0 \le i \le k-1$, the set $\overline{\mathcal{R}}(s)$ is the same. Since pt is a provided trace (i.e., there exists at least one such s, where a_{i+1} is enabled), so $a_{i+1} \in \overline{\mathcal{R}}(s)$ for all such s. This shows that all the provided traces are non-blockable, so all the traces are non-blockable too.

Second, we prove the direction from right to left by contraposition. We assume that C is not input-deterministic, so there exist two traces tr_1 and tr_2 with $\pi_1(tr_1)\lfloor_P = \pi_1(tr_2)\lfloor_P$ and $s_0 \overset{tr_1}{\Longrightarrow} s_1$, $s_0 \overset{tr_2}{\Longrightarrow} s_2$ such that $\overline{\mathcal{R}}(s_1) \neq \overline{\mathcal{R}}(s_2)$.

Without loss of generality, we assume that there is a provided event a such that $a \in \overline{\mathcal{R}}(s_1)$ and $a \notin \overline{\mathcal{R}}(s_2)$. Now $\pi_1(tr_1) \cdot \langle a \rangle$ is a provided trace of C that is blockable, which contradicts the assumption. □

We now present a procedure in Algorithm 1 that, given a component automaton C, constructs the interface publication automaton $\mathcal{I}(C)$. Each state of $\mathcal{I}(C)$ is a pair (Q, r) of a subset Q of states and a single state r of C. A pair (Q, r) is a state of $\mathcal{I}(C)$ if for some provided trace pt of C

- $s_0 \overset{pt}{\Longrightarrow} r$, and
- $Q = \{s \mid s_0 \overset{pt}{\Longrightarrow} s\}$.

Thus, in a tuple (Q, r), $r \in Q$ and the first element represents the set of all potentially reachable states for a given provided trace, whereas the second element is a specific reachable state for this trace. Notice that for the same r but

Algorithm 1. Construction of interface automaton $\mathcal{I}(C)$

```
Require: $C = (S, s_0, P, R, A, \delta)$
Ensure: $\mathcal{I}(C) = (S_I, (\{s_0\}, s_0), P, R, A, \delta_I)$, where $S_I \subseteq 2^S \times S$
1  initialization
2      $S_I := \{(\{s_0\}, s_0)\}$; $\delta_I := \emptyset$; $todo := \{(\{s_0\}, s_0)\}$; $done := \emptyset$
3  end initialization
4  while $todo \neq \emptyset$ do
5      choose $(Q, r) \in todo$; $todo := todo \setminus \{(Q, r)\}$; $done := done \cup \{(Q, r)\}$
6      foreach $a \in \bigcap_{s \in Q} \overline{\mathcal{R}}(s)$ do
7          $Q' := \bigcup_{s \in Q} \{s' \mid s \overset{a}{\Longrightarrow} s'\}$
8          foreach $(r \xrightarrow{e/E} r') \in \delta$ do
9              $\delta_I := \delta_I \cup \{(Q, r) \xrightarrow{e/E} (Q', r')\}$
10             if $(Q', r') \notin (todo \cup done)$ then
11                 $todo := todo \cup \{(Q', r')\}$
12                 $S_I := S_I \cup \{(Q', r')\}$
13     foreach $r \xrightarrow{\tau} r'$ with $r' \in Q$ do
14         $\delta_I := \delta_I \cup \{(Q, r) \xrightarrow{\tau} (Q, r')\}$
```

a provided trace pt_1 different from pt such that $s_0 \overset{pt_1}{\Longrightarrow} r$, there may be a different state (Q', r) for $\mathcal{I}(C)$. Then, the transition relation of $\mathcal{I}(C)$ is defined as $(Q, r) \xrightarrow{e/E} (Q', r')$ in $\mathcal{I}(C)$ if

- e is not a refusal at any state in Q, that is, $e \in \bigcap_{s \in Q} \overline{\mathcal{R}}(s)$,
- $r \xrightarrow{e/E} r'$, and
- $Q' = \{s' \mid \exists s \in Q \bullet s \overset{e}{\Longrightarrow} s'\}$.

From this it becomes clear that the first element Q of a compound state (Q, r) is necessary to compute if a transition step for an event e is possible at all, which is not the case if e can be refused in any state reachable by the same provided trace. The second element r is needed to identify the full transition step enabled in the state r; note that the required traces E in a compound event e/E can differ for all states in the set Q.

Algorithm 1 computes the pairs (Q, r) defined above and the transition relation to simulate the non-blockable executions of C. In the algorithm, the first elements of these state pairs, i.e., the sets $\{Q \mid \exists r \bullet (Q, r) \text{ is a state of } \mathcal{I}(C)\}$, result from a power-set construction similar to the construction of a deterministic automaton from a non-deterministic automaton. The variables *todo* and *done* are used to collect new reachable states that still have to be processed and states that have already been processed, respectively.

Example 11. In the internet connection component automaton given in Fig. 12, the provided trace $\langle login, read\rangle$ is non-blockable. However, $\langle login, print\rangle$ may

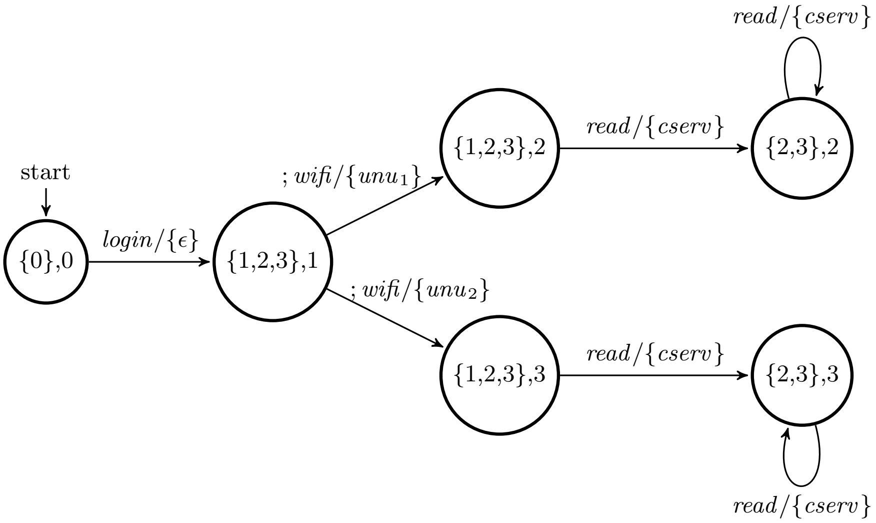


Fig. 15. Interface publication automaton of the internet connection component

be blocked during execution, because after *login* is called, the component may transit to state 3 at which *print* is not available. We use Algorithm 1 to generate the interface publication automaton in Fig. 15.

Three key correctness properties of the algorithm are stated in the following theorem.

Theorem 17 (Correctness of Algorithm 1). *The following properties hold for Algorithm 1.*

1. *For any given component automaton, the algorithm always terminates.*
2. *For any component C, $\mathcal{I}(C)$ is an input-deterministic automaton.*
3. $\mathcal{PP}(C) = \mathcal{PP}(\mathcal{I}(C))$ *and* $\mathcal{UT}(C) = \mathcal{UT}(\mathcal{I}(C))$.

The termination of the algorithm is obtained, because *todo* will eventually be empty: the set *done* increases for each iteration of the loop in the algorithm, and the union of *done* and *todo* is bounded. The proofs of the other two properties are given in [20].

It is interesting to study the states of $\mathcal{I}(C)$. Each of them is a pair (Q, r) of a set of a states of C and a state r. The state r and all states $s \in Q$ are target states of the same provided trace pt in C. Each transition from (Q, r) in $\mathcal{I}(C)$ adds a non-blockable event e to pt. Therefore, in the automaton $\mathcal{I}(C)$, $\overline{\mathcal{R}}((Q, r)) = \bigcap_{s \in Q} \overline{\mathcal{R}}(s)$, where $\overline{\mathcal{R}}(s)$ are non-blockable events of s in C. We simply write $\overline{\mathcal{R}}(Q, r)$. This means that each (Q, r) in $\mathcal{I}(C)$ actually encodes non-blockable events in an execution (called *global non-blockable events*) up to the state r in the execution of C. We also define the set of events that are enabled locally at a state s but globally refused (or blocked) as $\mathcal{B}(s) = \overline{\mathcal{R}}(s) \setminus \bigcup_{(Q,s) \in S_I} \overline{\mathcal{R}}(Q, s)$. This will be used in the definition of *alternative simulation* in the following subsection.

7.4 Composition and Refinement

The composition operations for component automata are derived from those for the component labeled transitions systems defined in Sect. 6. Further, component automata are special labeled transition systems and there is no data functionality (pre- and post-condition for the actions), and we do not have recursively defined component automata. Thus, we do not have to deal with divergent behavior caused by recursion. However, there are still possible livelock states in the dynamic behavior, but this can be characterized by interaction failures (similar to the CSP stable failure semantics [57]) with explicit consideration of livelocks. Another kind of execution failure is caused by cyclic method invocations of two interacting components. As discussed in Sect. 6, we use one dedicated state **error**. A transition that encounters cyclic method invocations will take the composite component to this state. In fact, **error** is a deadlock state in which no actions, including internal autonomous events are enabled. However, the difference of an **error** state from a normal deadlock state or a termination state is that the event e that leads from a state s to the **error** state, $s \xrightarrow{e}$ **error**, is disabled in state s too, and it should be eliminated from the traces of the system. Similarly, in a composition $K_1 \parallel K_2$, if a provided action in K_1 invokes a method provided by K_2 that is disabled, the transition also enters the **error** state. An automaton with the **error** state is depicted in Fig. 16. A general component automaton may also have the designated **error** state; an automaton with the **error** state that is not reachable from the initial state is equivalent to one without the **error** state.

More precisely, for an event $e \in \Omega$ and a state s of a component automaton C, if $s \xrightarrow{e}$ **error**, e is disabled in state s. Thus, for a general automaton with **error** state, the definition of the refusal events $\mathcal{R}(s)$ should take such an e into account. Then, Algorithm 1 also removes the transitions to **error**.

We now define two notions of failure sets for a component automaton.

Definition 35 (Failures sets of component automata). *Let C be a component automaton with the* **error** *state. A* **failure** *of C is a pair (tr, X) of a trace and a set of the events of Ω of C such that for every $e \in X$ there exists $s \in target(tr)$ such that $e \in \mathcal{R}(s)$, i.e., e is blocked in s, or s is a livelock state (i.e., an infinite sequence of internal steps is possible) and $X \subseteq \Omega$. We use $\mathcal{F}(C)$ to denote the set of failures of C.*

It can be shown that the set of traces $\mathcal{T}(C)$ and the set of provided traces $\mathcal{PT}(C)$ are given by

$$\mathcal{T}(C) = \{tr \mid \exists X \bullet (tr, X) \in \mathcal{F}(C)\}$$
$$\mathcal{PT}(C) = \{\pi_1(tr) \mid tr \in \mathcal{T}(C)\}$$

Analogously to the traces of open components defined in Sect. 4, for each provided trace pt, there is an associated set of sequences of required events,

$$\mathcal{RP}(pt) = \bigcup \{E_1 \cdots E_k \mid \exists tr \in \mathcal{T}(C) \bullet \pi_1(tr) = pt \wedge \pi_2(tr) = E_1 \cdots E_k\}.$$

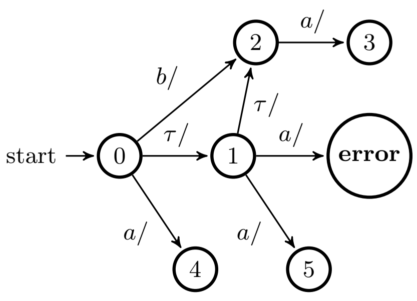


Fig. 16. An automaton with **error** state

The set of required traces for the non-blockable provided traces is given by $\mathcal{RP}(C) = \bigcup\{\mathcal{RP}(pt) \mid pt \in \mathcal{PP}(C)\}$. Now we formally define the refinement relation between component automata as the following partial order on their failures.

Definition 36 (Failure refinement of component automata). *A component automaton C_2 is a* **refinement** *of a component automaton C_1, denoted as $C_1 \sqsubseteq_f C_2$, if $\mathcal{F}(C_2) \subseteq \mathcal{F}(C_1)$.*

The properties of refinement between component automata, e.g., reflexivity and transitivity, are preserved by composition operations. However, we are interested in a "refinement relation" defined in terms of non-blocking provided and required traces.

Definition 37 (Alternative simulation). *A binary relation R over the set of states of a component automaton C is an* **alternative simulation** *if whenever $s_1 \; R \; s_2$,*

- *for any transition $s_1 \xrightarrow{e/E} s_1'$ with $e \in Act \cup \overline{\mathcal{R}}(s_1) \setminus \mathcal{B}(s_1)$ and* **error** $\notin$ *$autoR(s_1)$, there exist s_2' and E' such that $s_2 \xrightarrow{e/E'} s_2'$, where $E' \subseteq E$ and $s_1' \; R \; s_2'$;*
- *for any transition $s_2 \xrightarrow{e/E'} s_2'$ with $e \in Act \cup \overline{\mathcal{R}}(s_1) \setminus \mathcal{B}(s_1)$ and* **error** $\notin$ *$autoR(s_2)$, there exist s_1' and E such that $s_1 \xrightarrow{e/E} s_1'$, where $E' \subseteq E$ and $s_1' \; R \; s_2'$;*
- *$\mathcal{B}(s_2) \subseteq \mathcal{B}(s_1)$;*
- *if $s_2 \xrightarrow{e/}$* **error** *with $e \in Act \cup pIF$, then $s_1 \xrightarrow{e/}$* **error**.

We say that s_2 **alternative simulates** *s_1, written as $s_1 \lesssim s_2$, if there is an alternative simulation relation R such that $(s_1, s_2) \in R$. C_2 is an* **alternative refinement** *of C_1, denoted by $C_1 \sqsubseteq_{alt} C_2$, $C_1.init \lesssim C_2.init$, $C_1.pIF \subseteq C_2.pIF$ and $C_2.rIF \subseteq C_1.rIF$.*

The above definition is similar to the alternating simulation given in [16] and that is why we use the same term. But they are actually different. The main differences are (1) we only require a pair of states to keep the simulation relation with respect to the provided services that could not result in a deadlock; (2) we also require that a refinement should have smaller refusal sets at each location, which is similar to the stable failures model of CSP. Also notice that our refinement is not comparable with the failure refinement nor the failure-divergence refinement of CSP, because of the different requirements on the simulation of provided methods and required methods. However, if we do not suppose required methods, our definition is stronger than the failure refinement as well as the failure-divergence refinement.

The following theorem indicates that the component publication automaton constructed by Algorithm 1 is a refinement of the considered component automaton with respect to the above definition, which justifies that we can safely use the resulting component interface instead of the component at the interface level.

Theorem 18. *For any component automaton C, the refinement relation $C \sqsubseteq_{alt} \mathcal{I}(C)$ holds. If $C_1 \sqsubseteq_{alt} C_2$, then $\mathcal{I}(C_1) \sqsubseteq_{alt} \mathcal{I}(C_2)$.*

Proof. Let $R = \{(s, (Q, s)) \mid s \in S, (Q, s) \in S_I\}$. We show that R is a simulation relation.

For any $s\ R\ (Q, s)$,

- $s \xrightarrow{a/E} s'$ with $a \in \overline{\mathcal{R}}(s)$ and $a \notin \mathcal{B}(s)$. Then $a \in \mathcal{B}(Q, s)$ and $(Q, s) \xrightarrow{a/E} (Q', s')$.
- If $s \xrightarrow{;e/E} s'$ with $;e \in Act$, then $(Q, s) \xrightarrow{;e/E} (Q, s')$.
- For any $(Q, s) \xrightarrow{e/E} (Q', s')$ with $e \in Act \cup \overline{\mathcal{R}}(s) \setminus \mathcal{B}(Q, s)$, then $s \xrightarrow{e/E} s'$ and $s'\ R\ (Q', s')$.
- $\mathcal{B}(Q, s) \subseteq \mathcal{B}(s)$.

Hence, R is a simulation relation.

Now we prove the second part of the theorem. Let R_0 be a simulation for $C_1 \sqsubseteq_{alt} C_2$, then we show

$$R_0' = \left\{ ((Q_1, s_1), (Q_2, s_2)) \,\middle|\, \begin{array}{l} (s_1, s_2) \in R_0, \\ \forall r_1 \in Q_1 \exists r_2 \in Q_2 \bullet (r_1, r_2) \in R_0 \\ \forall r_2 \in Q_2 \exists r_1 \in Q_1 \bullet (r_1, r_2) \in R_0 \end{array} \right\}.$$

For any $(Q_1, s_1)\ R_0'\ (Q_2, s_2)$, $\overline{\mathcal{R}}(Q_1) \subseteq \overline{\mathcal{R}}(Q_2)$ and $\mathcal{B}(Q_1, s_1) = \mathcal{B}(Q_2, s_2) = \emptyset$. Then we can show that this is an alternative simulation between the initial states of $\mathcal{I}(C_1)$ and $\mathcal{I}(C_2)$. □

Theorem 19. *Given two component publication automata C_1 and C_2, if $C_1 \sqsubseteq_{alt} C_2$, then $\mathcal{PP}(C_1) \subseteq \mathcal{PP}(C_2)$, and for any non-blockable provided trace $pt \in \mathcal{PT}(C_1)$, $\mathcal{RP}(pt) \subseteq \mathcal{RP}(pt)$, where $\mathcal{RP}$ on the left is the set of required traces for pt in C_2 and $\mathcal{RP}$ on the right is that defined for C_1.*

This theorem can be proved by induction on the length of pt. The following theorem states that the refinement relation is preserved by the composition operator over component automata. We refer the reader to the paper [20] for its proof.

Theorem 20. *Given a component automaton C and two interface publication automata C_1 and C_2 such that $C_1 \sqsubseteq_{alt} C_2$, then $C_1 \otimes C \sqsubseteq_{alt} C_2 \otimes C$.*

Corollary 1. *Given two component interface automata C_1 and C_2, if $C_1 \sqsubseteq_{alt} C_2$, then $C_1 \parallel C \sqsubseteq_{alt} C_2 \parallel C$.*

The work on interface model in this section is a new development in rCOS. The results are still preliminary. There are still many open problems such as the relation between the failure refinement relation and the alternative refinement between component automata. A thorough study on the relation between CSP failure semantics theory and the automata simulation theory would deserve a Ph.D. thesis.

8 Conclusions

A major research objective of the rCOS method is to improve the scalability of semantic correctness preserving refinement between models in model-driven software engineering. The rCOS method promotes the idea that component-based software design is driven by model transformations in the front end, and verification and analysis techniques are integrated through the model transformations. It attacks the challenges of consistent integration of models for different viewpoints of a software system, for that different theories, techniques and tools can be applied effectively. The final goal of the integration is to support the separation of design concerns, those of the data functionality, interaction protocols and class structures in particular. rCOS provides a seamless combination of OO design and component-based design. As the semantic foundation presented in Sect. 2 and the models of components show, rCOS enables integration of classical specification and verification techniques, Hoare Logic and Predicate Transformers for data functionality, process algebras, finite state automata and temporal logics for reactive behavior. Refinement calculi for data functionality and reactive behavior are integrated as well.

In this chapter, we presented a model of component-based architecture, which is a generalization of the original rCOS components model presented in our early publications [10,13,25,35]. The semantics of the component architecture is based on unified labeled transition systems with a failure-divergence semantics and refinement for sequential, object-oriented, and reactive designs. Our semantics particularly integrates a data-based as well as an interaction-based view. This allowed us to introduce a general and unified model of components, which are the building blocks of a model-driven software architecture: primitive closed components, open components, as well as active and passive generalized components. The presented composition operators for parallel composition and operators for renaming, hiding, and plugging are used for the construction of complex systems out of predefined and refined components. Using refinement of components, this model is particularly suited for model transformations in model-driven software engineering [24,35,47]. Finally, a specific focus of this work was to study the interface behavior of components in order to being able to precisely specify contracts and publications for components, which enable the component's reusability in different contexts. This particularly included the computation of a component's provided protocol, which we identified as necessary to allow correct usage of a component in all desired situations.

Construction of models and model refinements are supported by the rCOS Modeler tool. The method has been tested on enterprise systems [11,12], remote medical systems [65] and service oriented systems [42].

Similar to JML [40], the rCOS method intends to convey the message that the design of a formal modeling notation can and should consider advanced features of architectural constructs in modern programming languages like Java. This will make it easier to use and understand for practical software engineers, who have difficulties to comprehend heavy mathematical constructs and operators.

The link of the rCOS models to classical semantic models is presented in this paper. We have not spent much space on related work as this has been discussed in previous papers, to which we have referred. However, we would like to emphasize the work on CSP and its failure-divergence theory [57] that motivated the input-deterministic interface model. Closely related models are Reo [14] and Circus [64]. The former model is related to the process model in rCOS, and Circus also deals with integration of data state into interaction behavior. In future work, we are interested in dealing with timing issues of components as another dimension of modeling. Also, with the separation of data functionality and flow of interaction control, we would like to investigate how the modeling method can be applied to workflow management, health care workflows in particular [4,21].

Acknowledgments. In this special occasion of a celebration of He Jifeng's 70th birthday, we would like to express our thanks for his contribution to the development of the semantic foundation of rCOS. Many of our former and current colleagues have made contributions to the development of the rCOS method and its tool support. Xiaoshan Li is another main contributor to the development of the rCOS theory, techniques and tool support. Jing Liu and her group have made significant contributions to the link of rCOS to UML and service oriented architecture; Xin Chen, Zhenbang Chen and Naijun Zhan to the component-based modeling and refinement; Charles Morisset, Xiaojian Liu, Shuling Wang, and Liang Zhao to the object-oriented semantics, refinement calculus and verification; Anders P. Ravn to the design of the tool and the CoCoME case study; Dan Li, Xiaoliang Wang, and Ling Yin to the tool development; Bin Lei and Cristiano Bertolini to testing techniques; and Martin Schäf to the automata-based model of interface behavior. The rCOS methods have been taught in many UNU-IIST training schools, inside and outside Macau, and we are grateful to the very helpful feedback and comments that we have received from the participants.

The work is supported by the projects GAVES, SAFEHR and PEARL funded by the Macau Science and Technology Development Fund, and the Chinese Natural Science Foundation grants NSFC-61103013, 91118007 and 60970031.

References

1. Abrial, J.R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press (1996)
2. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press (2010)
3. Back, R.J.R., von Wright, J.: Trace refinement of action systems. In: Jonsson, B., Parrow, J. (eds.) CONCUR 1994. LNCS, vol. 836, pp. 367–384. Springer, Heidelberg (1994)
4. Bertolini, C., Liu, Z., Schäf, M., Stolz, V.: Towards a formal integrated model of collaborative healthcare workflows. Tech. Rep. 450, IIST, United Nations University, Macao (2011), In: Liu, Z., Wassyng, A. (eds.) FHIES 2011. LNCS, vol. 7151, pp. 57–74. Springer, Heidelberg (2012)
5. Booch, G.: Object-Oriented Analysis and Design with Applications. Addison-Wesley, Boston (1994)
6. Brooks, F.P.: No silver bullet: Essence and accidents of software engineering. IEEE Computer 20(4), 10–19 (1987)
7. Brooks, F.P.: The mythical man-month: After 20 years. IEEE Software 12(5), 57–60 (1995)
8. Burstall, R., Goguen, J.: Putting theories together to make specifications. In: Reddy, R. (ed.) Proc. 5th Intl. Joint Conf. on Artificial Intelligence. Department of Computer Science, pp. 1045–1058. Carnegie-Mellon University, USA (1977)
9. Chandy, K.M., Misra, J.: Parallel Program Design: A Foundation. Addison-Wesley, Reading (1988)
10. 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), `http://www.iist.unu.edu/www/docs/techreports/reports/report350.pdf`
11. Chen, Z., et al.: Modelling with relational calculus of object and component systems - rCOS. In: Rausch, A., Reussner, R., Mirandola, R., Plášil, F. (eds.) The Common Component Modeling Example. LNCS, vol. 5153, pp. 116–145. Springer, Heidelberg (2008), `http://www.iist.unu.edu/www/docs/techreports/reports/report382.pdf`
12. Chen, Z., Liu, Z., Ravn, A.P., Stolz, V., Zhan, N.: Refinement and verification in component-based model driven design. Science of Computer Programming 74(4), 168–196 (2009), `http://www.sciencedirect.com/science/article/B6V17-4T9VP33-1/2/c4b7a123e06d33c2cef504862a5e54d5`
13. Chen, Z., Liu, Z., Stolz, V., Yang, L., Ravn, A.P.: A refinement driven component-based design. In: 12th Intl. Conf. on Engineering of Complex Computer Systems (ICECCS 2007), pp. 277–289. IEEE Computer Society (July 2007)
14. Clarke, D., Proença, J., Lazovik, A., Arbab, F.: Channel-based coordination via constraint satisfaction. Sci. Comput. Program. 76(8), 681–710 (2011)
15. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching-time temporal logic. In: Kozen, D. (ed.) Logic of Programs 1981. LNCS, vol. 131, pp. 52–71. Springer, Heidelberg (1982)

16. De Alfaro, L., Henzinger, T.: Interface automata. ACM SIGSOFT Software Engineering Notes 26(5), 109–120 (2001)
17. Dijkstra, E.W., Scholten, C.S.: Predicate Calculus and Program Semantics. Springer, New York (1990)
18. Dijkstra, E.W.: The humble programmer. Communications of the ACM 15(10), 859–866 (1972), an ACM Turing Award lecture
19. Dong, R., Faber, J., Liu, Z., Srba, J., Zhan, N., Zhu, J.: Unblockable compositions of software components. In: Grassi, V., Mirandola, R., Medvidovic, N., Larsson, M. (eds.) CBSE, pp. 103–108. ACM (2012)
20. Dong, R., Zhan, N., Zhao, L.: An interface model of software components. In: Zhu, H. (ed.) ICTAC 2013. LNCS, vol. 8049, pp. 157–174. Springer, Heidelberg (2013)
21. Faber, J.: A timed model for healthcare workflows based on csp. In: Breu, R., Hatcliff, J. (eds.) SEHC 2012, pp. 1–7. IEEE (2012) ISBN 978-1-4673-1843-3
22. Fischer, C.: Combination and Implementation of Processes and Data: from CSP-OZ to Java. Ph.D. thesis, University of Oldenburg (2000)
23. Fowler, M.: Refactoring – Improving the Design of Existing Code. Addison-Wesley, Menlo Park (1999)
24. He, J., Li, X., Liu, Z.: Component-based software engineering. In: Van Hung, D., Wirsing, M. (eds.) ICTAC 2005. LNCS, vol. 3722, pp. 70–95. Springer, Heidelberg (2005), `http://www.iist.unu.edu/www/docs/techreports/reports/report330.pdf`, uNU-IIST TR 330
25. He, J., Li, X., Liu, Z.: A theory of reactive components. Electr. Notes Theor. Comput. Sci. 160, 173–195 (2006)
26. He, J., Liu, Z., Li, X.: rCOS: A refinement calculus of object systems. Theoretical computer science 365(1-2), 109–142 (2006), `http://rcos.iist.unu.edu/publications/TCSpreprint.pdf`
27. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM 12(10), 576–580 (1969)
28. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, Upper Saddle River (1985)
29. Hoare, C.A.R., He, J.: Unifying Theories of Programming. Prentice-Hall, Upper Saddle River (1998)
30. Hoenicke, J., Olderog, E.R.: Combining specification techniques for processes, data and time. In: Butler, M., Petre, L., Sere, K. (eds.) IFM 2002. LNCS, vol. 2335, pp. 245–266. Springer, Heidelberg (2002), `http://link.springer.de/link/service/series/0558/bibs/2335/23350245.htm`
31. Holzmann, G.J.: The SPIN Model Checker: Primer and reference manual. Addison-Wesley (2004)
32. Holzmann, G.J.: Conquering complexity. IEEE Computer 40(12) (2007)
33. Johnson, J.: My Life Is Failure: 100 Things You Should Know to Be a Better Project Leader. Standish Group International, West Yarmouth (2006)
34. Jones, C.B.: Systematic Software Development using VDM. Prentice Hall, Upper Saddle River (1990)
35. Ke, W., Li, X., Liu, Z., Stolz, V.: rCOS: a formal model-driven engineering method for component-based software. Frontiers of Computer Science in China 6(1), 17–39 (2012)
36. 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)
37. Lamport, L.: The temporal logic of actions. ACM Transactions on Programming Languages and Systems 16(3), 872–923 (1994)

38. Lamport, L.: Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley (2002)
39. Larman, C.: Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process, 3rd edn. Prentice-Hall (2005)
40. Leavens, G.T.: JML's rich, inherited specifications for behavioral subtypes. In: Liu, Z., Kleinberg, R.D. (eds.) ICFEM 2006. LNCS, vol. 4260, pp. 2–34. Springer, Heidelberg (2006)
41. Leveson, N.G., Turner, C.S.: An investigation of the Therac-25 accidents. IEEE Computer 26(7), 18–41 (1993)
42. Liu, J., He, J.: Reactive component based service-oriented design – a case study. In: Proceedings of 11th IEEE International Conference on Engineering of Complex Computer Systems, pp. 27–36. IEEE Computer Society (2006)
43. Liu, Z.: Software development with UML. Tech. Rep. 259, IIST, United Nations University, P.O. Box 3058, Macao (2002)
44. Liu, Z., Joseph, M.: Specification and verification of fault-tolerance, timing, and scheduling. ACM Transactions on Programming Languages and Systems 21(1), 46–89 (1999)
45. Liu, Z., Kang, E., Zhan, N.: Composition and refinement of components. In: Butterfield, A. (ed.) Post Event Proceedings of UTP 2008. Lecture Notes in Computer Science vol. 5713. Springer, Berlin (2009)
46. Liu, Z., Mencl, V., Ravn, A.P., Yang, L.: Harnessing theories for tool support. In: Proc. of the Second Intl. Symp. on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2006), pp. 371–382. IEEE Computer Society (August 2006), `http://www.iist.unu.edu/www/docs/techreports/reports/report343.pdf`; full version as UNU-IIST Technical Report 343
47. 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), `http://www.iist.unu.edu/www/docs/techreports/reports/report406.pdf`, keynote, UNU-IIST TR 406
48. Lynch, N.A., Tuttle, M.R.: An introduction to input/output automata. CWI Quarterly 2(3), 219–246 (1989)
49. Manna, Z., Pnueli, A.: The temporal logic of reactive and concurrent systems:specification. Springer (1992)
50. Milner, R.: Communication and concurrency. Prentice-Hall, Inc., Upper Saddle River (1989)
51. Nielson, H., Nielson, F.: Semantics with Applications. A formal Introduction. Wiley (1993)
52. Object Managment Group: Model driven architecture - a technical perspective (2001), document number ORMSC 2001-07-01
53. Peter, L.: The Peter Pyramid. William Morrow, New York (1986)
54. Plotkin, G.D.: The origins of structural operational semantics. Journal of Logic and Algebraic Programming 60(61), 3–15 (2004)
55. Queille, J.P., Sifakis, J.: Specification and verification of concurrent systems in CESAR. In: Dezani-Ciancaglini, M., Montanari, U. (eds.) Programming 1982. LNCS, vol. 137, pp. 337–351. Springer, Heidelberg (1982)
56. Robinson, K.: Ariane 5: Flight 501 failure—a case study (2011), `http://www.cse.unsw.edu.au/~se4921/PDF/ariane5-article.pdf`
57. Roscoe, A.W.: Theory and Practice of Concurrency. Prentice-Hall, Upper Saddle River (1997)
58. Spivey, J.M.: The Z Notation: A Reference Manual, 2nd edn. Prentice Hall, Upper Saddle River (1992)

59. Stoy, J.E.: Denotational Semantics: The Scott-Strachey Approach to Programming Language Semantics. MIT Press, Cambridge (1977)
60. Szyperski, C.: Component Software: Beyond Object-Oriented Programming, 2nd edn. Addison-Wesley Longman Publishing Co., Inc., Boston (2002)
61. Vaandrager, F.W.: On the relationship between process algebra and input/output automata. In: LICS, pp. 387–398. IEEE Computer Society (1991)
62. Wang, Z., Wang, H., Zhan, N.: Refinement of models of software components. In: Shin, S.Y., Ossowski, S., Schumacher, M., Palakal, M.J., Hung, C.C. (eds.) SAC, pp. 2311–2318. ACM (2010)
63. Wirsing, M., Banâtre, J.-P., Hölzl, M., Rauschmayer, A. (eds.): Soft-Ware Intensive Systems. LNCS, vol. 5380. Springer, Heidelberg (2008)
64. Woodcock, J., Cavalcanti, A.: The semantics of circus. In: Bert, D., Bowen, J.P., Henson, M.C., Robinson, K. (eds.) ZB 2002. LNCS, vol. 2272, pp. 184–203. Springer, Heidelberg (2002)
65. Xiong, X., Liu, J., Ding, Z.: Design and verification of a trustable medical system. In: Johnsen, E.B., Stolz, V. (eds.) Proceedings of 3rd International Workshop on Harnessing Theories for Tool Support in Software. Electronic Notes in Theoretical Computer Science, vol. 266, pp. 77–92. Elsevier (2010)
66. Zhao, L., Liu, X., Liu, Z., Qiu, Z.: Graph transformations for object-oriented refinement. Formal Aspects of Computing 21(1-2), 103–131 (2009)

Model-Based Verification, Optimization, Synthesis and Performance Evaluation of Real-Time Systems

Uli Fahrenberg[1], Kim G. Larsen[2,⋆], and Axel Legay[1]

[1] Irisa / INRIA Rennes, France
[2] Department of Computer Science, Aalborg University, Selma Lagerlöfs Vej 300, 9220 Aalborg Øst, Denmark
kgl@cs.aau.dk

Abstract. This article aims at providing a concise and precise *Travellers Guide*, *Phrase Book* or *Reference Manual* to the timed automata modeling formalism introduced by Alur and Dill [8, 9]. The paper gives comprehensive definitions of timed automata, priced (or weighted) timed automata, timed games, stochastic timed automata and highlights a number of results on associated decision problems related to model checking, equivalence checking, optimal scheduling, the existence of winning strategies, and then statistical model checking.

1 Introduction

The model of timed automata, introduced by Alur and Dill [8, 9], has by now established itself as a classical formalism for describing the behaviour of real-time systems. A number of important algorithmic problems has been shown decidable for it, including reachability, model checking and several behavioural equivalences and preorders.

By now, real-time model checking tools such as Uppaal [20, 83] and Kronos [40] are based on the timed automata formalism and on the substantial body of research on this model that has been targeted towards transforming the early results into practically efficient algorithms — *e.g.* [16, 17, 22, 24] — and data structures — *e.g.* [23, 80, 82].

The maturity of a tool like Uppaal is witnessed by the numerous applications — *e.g.* [50, 57, 68, 73, 78, 81, 86, 87] — to the verification of industrial case-studies spanning real-time controllers and real-time communication protocols. More recently, model-checking tools in general and Uppaal in particular have been applied to solve realistic scheduling problems by a reformulation as reachability problems — *e.g.* [1, 65, 72, 89].

Aiming at providing methods for performance analysis, a recent extension of timed automata is that of *priced* or *weighted* timed automata [10, 21], which

⋆ Corresponding author.

makes it possible to formulate and solve *optimal* scheduling problems. Surprisingly, a number of properties have been shown to be decidable for this formalism [10, 21, 34, 59, 84]. The recently developed UPPAAL-CORA tool provides an efficient tool for solving cost-optimal reachability problems [79] and has been applied successfully to a number of optimal scheduling problems, *e.g.* [18,25,67].

Most recently, substantial efforts have been made on the automatic synthesis of (correct-by-construction) controllers from timed games for given control objectives. From early decidability results [13, 91] the effort has lead to efficient on-the-fly algorithms [44, 102] with the newest of the UPPAAL toolset, UPPAAL-TIGA [19], UPPAAL-SMC [53, 54], providing an efficient tool implementation with industrial applications emerging, *e.g.* [75].

This survey paper aims at providing a concise and precise *Travellers Guide*, *Phrase Book* or *Reference Manual* to the land and language of timed automata. The article gives comprehensive definitions of timed automata, weighted timed automata, and timed games and highlights a number of results on associated decision problems related to model checking, equivalence checking, optimal scheduling, the existence of winning strategies, and statistical model checking. The intention is that the paper should provide an easy-to-access collection of important results and overview of the field to anyone interested.

We acknowledge the assistance of Claus Thrane who has been a co-author on previous editions of this survey [61, 62, 64]. We would also like to thank the students of the Marktoberdorf and Quantitative Model Checking PhD schools for their useful comments and help in weeding out a number of errors in previous editions, as well as an anonymous reviewer who provided many useful remarks for the invited paper [63] at FSEN 2009.

2 Timed Automata

In this section we review the notion of timed automata introduced by Alur and Dill [8, 9] as a formalism for describing the behaviour of real-time systems. We review the syntax and semantics and highlight the, by now classical, region construction underlying the decidability of several associated problems.

Here we illustrate how regions are applied in showing decidability of reachability and timed and untimed (bi)similarity. However, the notion of region does not provide the means for efficient tool implementations. The verification engine of UPPAAL instead applies so-called zones, which are *convex unions* of regions. We give a brief account of zones as well as their efficient representation and manipulation using difference-bound matrices.

2.1 Syntax and Semantics

Definition 1. *The set $\Phi(C)$ of* clock constraints *φ over a finite set (of* clocks*) C is defined by the grammar*

$$\varphi ::= x \bowtie k \mid \varphi_1 \wedge \varphi_2 \qquad (x \in C, k \in \mathbb{Z}, \bowtie \in \{\le, <, \ge, >\}).$$

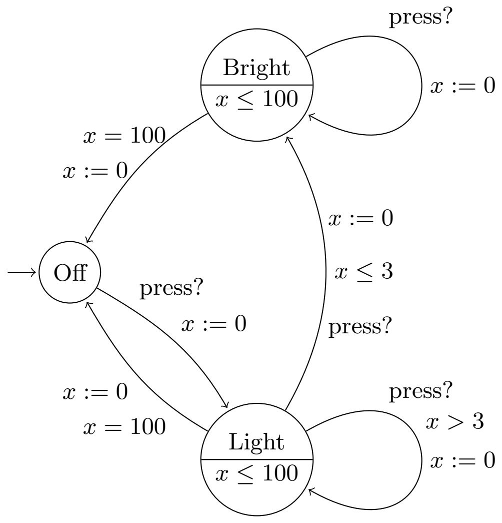


Fig. 1. A light switch modelled as a timed automaton

The set $\Phi^+(C)$ *of* extended clock constraints φ *is defined by the grammar*

$$\varphi ::= x \bowtie k \mid x - y \bowtie k \mid \varphi_1 \wedge \varphi_2 \qquad (x, y \in C, k \in \mathbb{Z}, \bowtie \in \{\leq, <, \geq, >\}).$$

Remark 1. The clock constraints in $\Phi(C)$ above are also called *diagonal-free* clock constraints, and the additional ones in $\Phi^+(C)$ are called *diagonal.* We restrict ourselves to diagonal-free clock constraints here; see Remark 4 for one reason. For additional modelling power, timed automata with diagonal constraints can be used, as it is shown in [9,29] that any such automaton can be converted to a diagonal-free one; however the conversion may lead to an exponential blow-up.

Definition 2. *A* timed automaton *is a tuple* $(L, \ell_0, F, C, \Sigma, I, E)$ *consisting of a finite set* L *of locations, an initial location* $\ell_0 \in L$*, a set* $F \subseteq L$ *of final locations, a finite set* C *of clocks, a finite set* Σ *of actions, a location invariants mapping* $I : L \to \Phi(C)$*, and a set* $E \subseteq L \times \Phi(C) \times \Sigma \times 2^C \times L$ *of edges.*

Here 2^C denotes the set of subsets (*i.e.* the power set) of C. We shall write $\ell \xrightarrow{\varphi,a,r} \ell'$ for an edge $(\ell, \varphi, a, r, \ell') \in E$. In figures, resets are written as assignments to zero, *e.g.* $x := 0$.

Example 1. Figure 1 provides a timed automaton model of an intelligent light switch. Starting in the "Off" state, a press of the button turns the light on, and it remains in this state for 100 time units (*i.e.* until clock $x = 100$), at which time the light turns off again. During this time, an additional press resets the clock x

and prolongs the time in the state by 100 time units. Pressing the button twice, with at most three time units between the presses, triggers a special bright light.

Definition 3. *A* clock valuation *on a finite set C of clocks is a mapping $v : C \to \mathbb{R}_{\geq 0}$. The* initial *valuation v_0 is given by $v_0(x) = 0$ for all $x \in C$. For a valuation v, $d \in \mathbb{R}_{\geq 0}$, and $r \subseteq C$, the valuations $v + d$ and $v[r]$ are defined by*

$$(v + d)(x) = v(x) + d$$
$$v[r](x) = \begin{cases} 0 & \text{for } x \in r, \\ v(x) & \text{for } x \notin r. \end{cases}$$

Extending the notation for power set introduced above, we will in general write B^A for the set of mappings from a set A to a set B. The set of clock valuations on C is thus $\mathbb{R}_{\geq 0}^C$.

Definition 4. *The* zone *of an extended clock constraint in $\Phi^+(C)$ is the set of clock valuations $C \to \mathbb{R}_{\geq 0}$ given inductively by*

$$\begin{aligned} [\![x \bowtie k]\!] &= \{v : C \to \mathbb{R}_{\geq 0} \mid v(x) \bowtie k\}, \\ [\![x - y \bowtie k]\!] &= \{v : C \to \mathbb{R}_{\geq 0} \mid v(x) - v(y) \bowtie k\}, \text{ and} \\ [\![\varphi_1 \wedge \varphi_2]\!] &= [\![\varphi_1]\!] \cap [\![\varphi_2]\!]. \end{aligned}$$

We shall write $v \models \varphi$ instead of $v \in [\![\varphi]\!]$.

Definition 5. *The* semantics *of a timed automaton $A = (L, \ell_0, F, C, \Sigma, I, E)$ is the transition system $[\![A]\!] = (S, s_0, \Sigma \cup \mathbb{R}_{\geq 0}, T = T_s \cup T_d)$ given as follows:*

$$\begin{aligned} S &= \big\{(\ell, v) \in L \times \mathbb{R}_{\geq 0}^C \bigm| v \models I(\ell)\big\} \qquad\qquad s_0 = (\ell_0, v_0) \\ T_s &= \big\{(\ell, v) \xrightarrow{a} (\ell', v') \bigm| \exists \ell \xrightarrow{\varphi, a, r} \ell' \in E : v \models \varphi, v' = v[r]\big\} \\ T_d &= \big\{(\ell, v) \xrightarrow{d} (\ell, v + d) \bigm| \forall d' \in [0, d] : v + d' \models I(\ell)\big\} \end{aligned}$$

Remark 2. The transition system $[\![A]\!]$ from above is an example of what is known as a *timed transition system*, *i.e.* a transition system where the label set includes $\mathbb{R}_{\geq 0}$ as a subset and which satisfies certain additivity and time determinacy properties. We refer to [2] for a more in-depth treatment.

Also note that the semantics $[\![A]\!]$ contains no information about final states (derived from the final locations in F); this is mostly for notational convenience.

Definition 6. *A (finite)* run *of a timed automaton $A = (L, \ell_0, F, C, \Sigma, I, E)$ is a finite path $\rho = (\ell_0, v_0) \to \cdots \to (\ell_k, v_k)$ in $[\![A]\!]$. It is said to be* accepting *if $\ell_k \in F$.*

Example 1 (continued). The light switch model from figure 1 has as state set

$$S = \{\text{Off}\} \times \mathbb{R}_{\geq 0} \cup \{\text{Light}, \text{Bright}\} \times [0, 100]$$

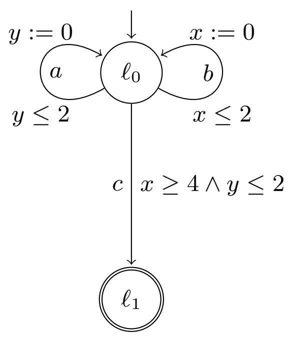


Fig. 2. A timed automaton with two clocks

where we identify valuations with their values at x. A few example runs are given below; we abbreviate "press?" to "p":

$$(\text{Off}, 0) \xrightarrow{150} (\text{Off}, 150) \xrightarrow{\text{p}} (\text{Light}, 0) \xrightarrow{100} (\text{Light}, 100) \to (\text{Off}, 0)$$

$$(\text{Off}, 0) \xrightarrow{\text{p}} (\text{Light}, 0) \xrightarrow{10} (\text{Light}, 10) \xrightarrow{\text{p}} (\text{Light}, 0) \xrightarrow{100} (\text{Light}, 100) \to (\text{Off}, 0)$$

$$(\text{Off}, 0) \xrightarrow{\text{p}} (\text{Light}, 0) \xrightarrow{1} (\text{Light}, 1) \xrightarrow{\text{p}} (\text{Bright}, 0) \xrightarrow{100} (\text{Bright}, 100) \to (\text{Off}, 0)$$

2.2 Reachability

We are concerned with the following problem: Given a timed automaton $A = (L, \ell_0, F, C, \Sigma, I, E)$, is any of the locations in F reachable? We shall later define the *timed language* generated by a timed automaton and see that this reachability problem is equivalent to *emptiness checking*: Is the timed language generated by A non-empty?

Example 2 (cf. [2, Ex. 11.7]). Figure 2 shows a timed automaton A with two clocks and a final location ℓ_1. To ask whether ℓ_1 is reachable amounts for this automaton to the question whether there is a finite sequence of a- and b-transitions from ℓ_0 which brings clock values into accordance with the guard $x \geq 4 \land y \leq 2$ on the edge leading to ℓ_1.

An immediate obstacle to reachability checking is the infinity of the state space of A. In general, the transition system $[\![A]\!]$ has uncountably many states, hence straight-forward reachability algorithms do not work for us.

Notation 1. *The* derived transition relations *in a timed automaton $A = (L, \ell_0, F, C, \Sigma, I, E)$ are defined as follows: For (ℓ, v), (ℓ', v') states in $[\![A]\!]$, we say that*

- *$(\ell, v) \xrightarrow{\delta} (\ell', v')$ if $(\ell, v) \xrightarrow{d} (\ell', v')$ in $[\![A]\!]$ for some $d > 0$,*
- *$(\ell, v) \xrightarrow{\alpha} (\ell', v')$ if $(\ell, v) \xrightarrow{a} (\ell', v')$ in $[\![A]\!]$ for some $a \in \Sigma$, and*
- *$(\ell, v) \rightsquigarrow (\ell', v')$ if $(\ell, v) (\xrightarrow{\delta} \cup \xrightarrow{\alpha})^* (\ell', v')$.*

Definition 7. *The set of* reachable locations *in a timed automaton* $A = (L, \ell_0, F, C, \Sigma, I, E)$ *is*

$$\text{Reach}(A) = \big\{\ell \in L \;\big|\; \exists v : C \to \mathbb{R}_{\geq 0} : (\ell_0, v_0) \rightsquigarrow (\ell, v)\big\}.$$

Hence we can now state the reachability problem as follows:

Problem 1 (Reachability). Given a timed automaton $A = (L, \ell_0, F, C, \Sigma, I, E)$, is $\text{Reach}(A) \cap F \neq \emptyset$?

Definition 8. *Let* $A = (L, \ell_0, F, C, \Sigma, I, E)$ *be a timed automaton. A relation* $R \subseteq L \times \mathbb{R}_{\geq 0}^C \times L \times \mathbb{R}_{\geq 0}^C$ *is a* time-abstracted simulation *provided that for all* $(\ell_1, v_1)\ R\ (\ell_2, v_2)$,

- *for all* $(\ell_1, v_1) \xrightarrow{\delta} (\ell_1', v_1')$ *there exists some* (ℓ_2', v_2') *such that* $(\ell_1', v_1')\ R\ (\ell_2', v_2')$ *and* $(\ell_2, v_2) \xrightarrow{\delta} (\ell_2', v_2')$, *and*
- *for all* $a \in \Sigma$ *and* $(\ell_1, v_1) \xrightarrow{a} (\ell_1', v_1')$, *there exists some* (ℓ_2', v_2') *such that* $(\ell_1', v_1')\ R\ (\ell_2', v_2')$ *and* $(\ell_2, v_2) \xrightarrow{a} (\ell_2', v_2')$.

*R is said to be F-*sensitive *if additionally,* $(\ell_1, v_1)\ R\ (\ell_2, v_2)$ *implies that* $\ell_1 \in F$ *if and only if* $\ell_2 \in F$. *A* time-abstracted bisimulation *is a time-abstracted simulation which is also symmetric; we write* $(\ell_1, v_1) \approx (\ell_2, v_2)$ *whenever* $(\ell_1, v_1)\ R\ (\ell_2, v_2)$ *for a time-abstracted bisimulation R.*

Note that $\approx$ is itself a time-abstracted bisimulation, which is easily shown to be an equivalence relation and hence symmetric, reflexive and transitive. Observe also that a time-abstracted (bi)simulation on A is the same as a standard (bi)simulation on the transition system derived from $[\![A]\!]$ with transitions $\xrightarrow{\delta}$ and $\xrightarrow{a}$. Likewise, the quotient introduced below is just the standard bisimulation quotient of this derived transition system.

Definition 9. *Let* $A = (L, \ell_0, F, C, \Sigma, I, E)$ *be a timed automaton and* $R \subseteq L \times \mathbb{R}_{\geq 0}^C \times L \times \mathbb{R}_{\geq 0}^C$ *a time-abstracted bisimulation which is also an equivalence. The* quotient *of* $[\![A]\!] = (S, s_0, \Sigma \cup \mathbb{R}_{\geq 0}, T)$ *with respect to R is the transition system* $[\![A]\!]_R = (S_R, s_R^0, \Sigma \cup \{\delta\}, T_R)$ *given by* $S_R = S/R$, $s_R^0 = [s_0]_R$, *and with transitions*

- $\pi \xrightarrow{\delta} \pi'$ *whenever* $(\ell, v) \xrightarrow{\delta} (\ell', v')$ *for some* $(\ell, v) \in \pi$, $(\ell', v') \in \pi'$, *and*
- $\pi \xrightarrow{a} \pi'$ *whenever* $(\ell, v) \xrightarrow{a} (\ell', v')$ *for some* $(\ell, v) \in \pi$, $(\ell', v') \in \pi'$.

The following proposition expresses that F-sensitive quotients are sound and complete with respect to reachability.

Proposition 1 ([5]). *Let* $A = (L, \ell_0, F, C, \Sigma, I, E)$ *be a timed automaton,* $R \subseteq L \times \mathbb{R}_{\geq 0}^C \times L \times \mathbb{R}_{\geq 0}^C$ *an F-sensitive time-abstracted bisimulation and* $\ell \in F$. *Then* $\ell \in \text{Reach}(A)$ *if and only if there is a reachable state π in* $[\![A]\!]_R$ *and* $v : C \to \mathbb{R}_{\geq 0}$ *such that* $(\ell, v) \in \pi$.

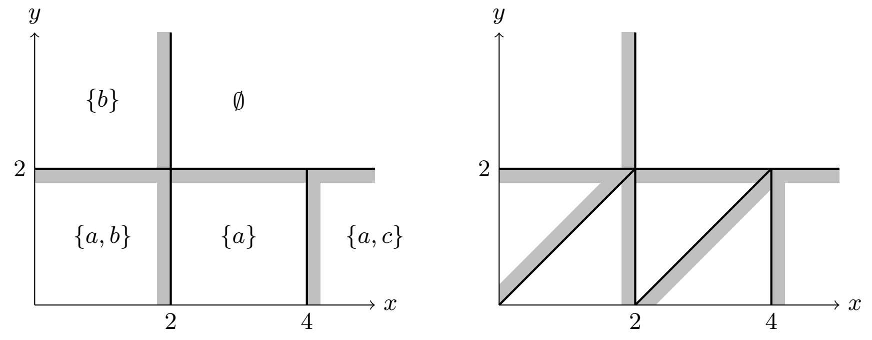


Fig. 3. Time-abstracted bisimulation classes for the two-clock timed automaton from Example 2. Left: equivalence classes for switch transitions only; right: equivalence classes for switch and delay transitions.

Example 2 (continued). We shall now try to construct, in a naïve way, a time-abstracted bisimulation R, which is as coarse as possible, for the timed automaton A from Figure 2. Note first that we cannot have $(\ell_0, v)\ R\ (\ell_1, v')$ for any $v, v' : C \to \mathbb{R}_{\geq 0}$ because $\ell_1 \in F$ and $\ell_0 \notin F$. On the other hand it is easy to see that we can let $(\ell_1, v)\ R\ (\ell_1, v')$ for all $v, v' : C \to \mathbb{R}_{\geq 0}$, which leaves us with constructing R on the states involving ℓ_0.

We handle switch transitions $\xrightarrow{\alpha}$ first: If $v, v' : C \to \mathbb{R}_{\geq 0}$ are such that $v(y) \leq 2$ and $v'(y) > 2$, the state (ℓ_0, v) has an a-transition available while the state (ℓ_0, v') has not, hence these cannot be related in R. Similarly we have to distinguish states (ℓ_0, v) from states (ℓ_0, v') where $v(x) \leq 2$ and $v'(x) > 2$ because of b-transitions, and states (ℓ_0, v) from states (ℓ_0, v') where $v(x) < 4$ and $v'(x) \geq 4$ because of c-transitions. Altogether this gives the five classes depicted to the left of Figure 3, where the shading indicates to which class the boundary belongs, and we have written the set of available actions in the classes.

When also taking delay transitions $\xrightarrow{\delta}$ into account, one has to partition the state space further: From a valuation v in the class marked $\{a, b\}$ in the left of the figure, a valuation in the class marked $\{a\}$ can only be reached by a delay transition if $v(y) < v(x)$; likewise, from the $\{a\}$ class, the $\{a, c\}$ class can only be reached if $v(y) \leq v(x) - 2$. Hence these two classes need to be partitioned as shown to the right of Figure 3.

It can easily be shown that no further partitioning is needed, thus we have defined the coarsest time-abstracted bisimulation relation for A, altogether with eight equivalence classes.

2.3 Regions

Motivated by the construction in the example above, we now introduce a time-abstracted bisimulation with a *finite quotient*. To ensure finiteness, we need the maximal constants to which respective clocks are compared in the invariants and guards of a given timed automaton. These may be defined as follows.

Definition 10. *For a finite set C of clocks, the* maximal constant *mapping $c_{\max} : C \to \mathbb{Z}^{\Phi(C)}$ is defined inductively as follows:*

$$c_{\max}(x)(y \bowtie k) = \begin{cases} k & \text{if } y = x \\ 0 & \text{if } y \neq x \end{cases}$$
$$c_{\max}(x)(\varphi_1 \wedge \varphi_2) = \max\big(c(x)(\varphi_1), c(x)(\varphi_2)\big)$$

For a timed automaton $A = (L, \ell_0, F, C, \Sigma, I, E)$, the maximal constant mapping is $c_A : C \to \mathbb{Z}$ defined by

$$c_A(x) = \max\big\{c_{\max}(x)(I(\ell)), c_{\max}(x)(\varphi) \;\big|\; \ell \in L, \ell \xrightarrow{\varphi,a,r} \ell' \in E\big\}.$$

Notation 2. *For $d \in \mathbb{R}_{\geq 0}$ we write $\lfloor d \rfloor$ and $\langle d \rangle$ for the integral, respectively fractional, part of d, so that $d = \lfloor d \rfloor + \langle d \rangle$.*

Definition 11. *For a timed automaton $A = (L, \ell_0, F, C, \Sigma, I, E)$, valuations $v, v' : C \to \mathbb{R}_{\geq 0}$ are said to be* region equivalent*, denoted $v \cong v'$, if*

- *$\lfloor v(x) \rfloor = \lfloor v'(x) \rfloor$ or $v(x), v'(x) > c_A(x)$, for all $x \in C$, and*
- *$\langle v(x) \rangle = 0$ iff $\langle v'(x) \rangle = 0$, for all $x \in C$, and*
- *$\langle v(x) \rangle \leq \langle v(y) \rangle$ iff $\langle v'(x) \rangle \leq \langle v'(y) \rangle$ for all $x, y \in C$.*

Proposition 2 ([5]). *For a timed automaton $A = (L, \ell_0, F, C, \Sigma, I, E)$, the equivalence relation $\cong$ defined on states of $[\![A]\!]$ by $(\ell, v) \cong (\ell', v')$ if $\ell = \ell'$ and $v \cong v'$ is an F-sensitive time-abstracted bisimulation. The quotient $[\![A]\!]_{\cong}$ is finite.*

The equivalence classes of valuations of A with respect to $\cong$ are called *regions*, and the quotient $[\![A]\!]_{\cong}$ is called the *region automaton* associated with A.

Proposition 3 ([9]). *The number of regions for a timed automaton A with a set C of n clocks is bounded above by*

$$n! \cdot 2^n \cdot \prod_{x \in C} (2c_A(x) + 2).$$

Example 2 (continued). The 69 regions of the timed automaton A from Figure 2 are depicted in Figure 4.

Propositions 1 and 2 together now give the decidability part of the theorem below; for PSPACE-completeness see [7, 49].

Theorem 3. *The reachability problem for timed automata is* PSPACE*-complete.*

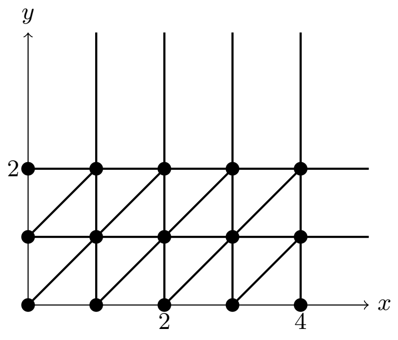


Fig. 4. Clock regions for the timed automaton from Example 2

2.4 Behavioural Refinement Relations

We have already introduced time-abstracted simulations and bisimulations in Definition 8. As a corollary of Proposition 2, these are decidable:

Theorem 4. *Time-abstracted simulation and bisimulation are decidable for timed automata.*

Proof. One only needs to see that time-abstracted (bi)simulation in the timed automaton is the same as ordinary (bi)simulation in the associated region automaton; indeed, any state in $[\![A]\!]$ is untimed bisimilar to its image in $[\![A]\!]_{\cong}$. The result follows by finiteness of the region automaton. □

The following provides a *time-sensitive* variant of (bi)simulation.

Definition 12. *Let $A = (L, \ell_0, F, C, \Sigma, I, E)$ be a timed automaton. A relation $R \subseteq L \times \mathbb{R}_{\geq 0}^C \times L \times \mathbb{R}_{\geq 0}^C$ is a* timed simulation *provided that for all $(\ell_1, v_1)\ R\ (\ell_2, v_2)$,*

- *for all $(\ell_1, v_1) \xrightarrow{d} (\ell_1', v_1')$, $d \in \mathbb{R}_{\geq 0}$, there exists some (ℓ_2', v_2') such that $(\ell_1', v_1')\ R\ (\ell_2', v_2')$ and $(\ell_2, v_2) \xrightarrow{d} (\ell_2', v_2')$, and*
- *for all $(\ell_1, v_1) \xrightarrow{a} (\ell_1', v_1')$, $a \in \Sigma$, there exists some (ℓ_2', v_2') such that $(\ell_1', v_1')\ R\ (\ell_2', v_2')$ and $(\ell_2, v_2) \xrightarrow{a} (\ell_2', v_2')$.*

A timed bisimulation *is a timed simulation which is also symmetric, and two states (ℓ_1, v_1), $(\ell_2, v_2) \in [\![A]\!]$ are said to be* timed bisimilar*, written $(\ell_1, v_1) \sim (\ell_2, v_2)$, if there exists a timed bisimulation R for which $(\ell_1, v_1)\ R\ (\ell_2, v_2)$.*

Note that $\sim$ is itself a timed bisimulation on A, which is easily shown to be an equivalence relation and hence transitive, reflexive and symmetric.

Definition 13. *Two timed automata $A = (L^A, \ell_0^A, F^A, C^A, \Sigma^A, I^A, E^A)$ and $B = (L^B, \ell_0^B, F^B, C^B, \Sigma^B, I^B, E^B)$ are said to be* timed bisimilar*, denoted $A \sim B$, if $(\ell_0^A, v_0) \sim (\ell_0^B, v_0)$ in the disjoint-union transition system $[\![A]\!] \sqcup [\![B]\!]$.*

Timed simulation of timed automata can be analogously defined. The following decidability result was established for *parallel timed processes* in [46]; below we give a version of the proof which has been adapted for timed automata. Later, in Section 4 on page 89, we shall give an alternative proof which uses timed *games*.

Theorem 5. *Timed similarity and bisimilarity are decidable for timed automata.*

Before the proof, we need a few auxiliary definitions and lemmas. The first is a product of timed transition systems which synchronizes on time, but not on actions:

Definition 14. *The* independent product *of the timed transition systems* $[\![A]\!] = (S^A, s_0^A, \Sigma^A \cup \mathbb{R}_{\geq 0}, T^A)$, $[\![B]\!] = (S^B, s_0^B, \Sigma^B \cup \mathbb{R}_{\geq 0}, T^B)$ *associated with timed automata A, B is* $[\![A]\!] \times [\![B]\!] = (S, s_0, \Sigma^A \cup \Sigma^B \cup \mathbb{R}_{\geq 0}, T)$ *given by*

$$\begin{aligned} S &= S^A \times S^B \qquad\qquad s_0 = (s_0^A, s_0^B) \\ T &= \big\{(p,q) \xrightarrow{a} (p',q) \mid a \in \Sigma, p \xrightarrow{a} p' \in T^A\big\} \\ &\quad \cup \big\{(p,q) \xrightarrow{b} (p,q') \mid b \in \Sigma, q \xrightarrow{b} q' \in T^B\big\} \\ &\quad \cup \big\{(p,q) \xrightarrow{d} (p',q') \mid d \in \mathbb{R}_{\geq 0}, p \xrightarrow{d} p' \in T^A, q \xrightarrow{d} q' \in T^B\big\} \end{aligned}$$

We need to extend region equivalence $\cong$ to the independent product. Below, $\oplus$ denotes vector concatenation (direct sum); note that $(p_1, q_1) \cong (p_2, q_2)$ is not the same as $p_1 \cong p_2$ and $q_1 \cong q_2$, as fractional orderings $\langle x^A \rangle \bowtie \langle x^B \rangle$, for $x^A \in C^A$, $x^B \in C^B$, have to be accounted for in the former, but not in the latter. Hence $(p_1, q_1) \cong (p_2, q_2)$ implies $p_1 \cong p_2$ and $q_1 \cong q_2$, but not vice-versa.

Definition 15. *For states $p_i = (\ell^{p_i}, v^{p_i})$ in $[\![A]\!]$ and $q_i = (\ell^{q_i}, v^{q_i})$ in $[\![B]\!]$ for $i = 1, 2$, we say that $(p_1, q_1) \cong (p_2, q_2)$ iff $\ell^{p_1} = \ell^{p_2} \wedge \ell^{q_1} = \ell^{q_2}$ and $v^{p_1} \oplus v^{q_1} \cong v^{p_2} \oplus v^{q_2}$.*

Note that the number of states in $\big([\![A]\!] \times [\![B]\!]\big)_{\cong}$ is finite, with an upper bound given by Proposition 3. Next we define transitions in $\big([\![A]\!] \times [\![B]\!]\big)_{\cong}$:

Notation 6. *Regions in $\big([\![A]\!] \times [\![B]\!]\big)_{\cong}$ will be denoted X, X'. The equivalence class of a pair $(p,q) \in [\![A]\!] \times [\![B]\!]$ is denoted $[p,q]$.*

Definition 16. *For $X, X' \in \big([\![A]\!] \times [\![B]\!]\big)_{\cong}$ we say that*

- *$X \xrightarrow{a}_\ell X'$ for $a \in \Sigma$ if for all $(p,q) \in X$ there exists $(p',q) \in X'$ such that $(p,q) \xrightarrow{a} (p',q)$ in $[\![A]\!] \times [\![B]\!]$,*
- *$X \xrightarrow{b}_r X'$ for $b \in \Sigma$ if for all $(p,q) \in X$ there exists $(p,q') \in X'$ such that $(p,q) \xrightarrow{b} (p,q')$ in $[\![A]\!] \times [\![B]\!]$, and*
- *$X \xrightarrow{\delta} X'$ if for all $(p,q) \in X$ there exists $d \in \mathbb{R}_{\geq 0}$ and $(p',q') \in X'$ such that $(p,q) \xrightarrow{d} (p',q')$.*

Definition 17. *A subset* $\mathcal{B} \subseteq \big([\![A]\!] \times [\![B]\!]\big)_{\cong}$ *is a* symbolic bisimulation *provided that for all* $X \in \mathcal{B}$*,*

– *whenever* $X \xrightarrow{a}_{\ell} X'$ *for some* $X' \in \big([\![A]\!] \times [\![B]\!]\big)_{\cong}$*, then* $X' \xrightarrow{a}_{r} X''$ *for some* $X'' \in \mathcal{B}$*,*
– *whenever* $X \xrightarrow{a}_{r} X'$ *for some* $X' \in \big([\![A]\!] \times [\![B]\!]\big)_{\cong}$*, then* $X' \xrightarrow{a}_{\ell} X''$ *for some* $X'' \in \mathcal{B}$*, and*
– *whenever* $X \xrightarrow{\delta} X'$ *for some* $X' \in \big([\![A]\!] \times [\![B]\!]\big)_{\cong}$*, then* $X' \in \mathcal{B}$*.*

Note that it is decidable whether $\big([\![A]\!] \times [\![B]\!]\big)_{\cong}$ admits a symbolic bisimulation. The following proposition finishes the proof of Theorem 5.

Proposition 4. *The quotient* $\big([\![A]\!] \times [\![B]\!]\big)_{\cong}$ *admits a symbolic bisimulation if and only if* $A \sim B$*.*

Proof (cf. [46]). For a given symbolic bisimulation $\mathcal{B} \subseteq \big([\![A]\!] \times [\![B]\!]\big)_{\cong}$, the set $R_{\mathcal{B}} = \big\{(p,q) \bigm| [p,q] \in \mathcal{B}\big\} \subseteq [\![A]\!] \times [\![B]\!]$ is a timed bisimulation. For the other direction, one can construct a symbolic bisimulation from a timed bisimulation $R \subseteq [\![A]\!] \times [\![B]\!]$ by $\mathcal{B}_R = \big\{[p,q] \bigm| (p,q) \in R\big\}$. □

2.5 Language Inclusion and Equivalence

Similarly to the untimed setting, there is also a notion of language inclusion and equivalence for timed automata. We need to introduce the notion of *timed trace* first. Note that we restrict to *finite* timed traces here; similar results are available for infinite traces in timed automata with Büchi or Muller acceptance conditions, see [9].

Definition 18. *A* timed trace *over a finite set of actions* Σ *is a finite sequence* $((t_1,a_1),(t_2,a_2),\dots,(t_k,a_k))$*, where* $a_i \in \Sigma$ *and* $t_i \in \mathbb{R}_{\geq 0}$ *for* $i = 1,\dots,k$*, and* $t_i < t_{i+1}$ *for* $i = 1,\dots,k-1$*. The set of all timed traces over* Σ *is denoted* $T\Sigma^*$*.*

In a pair (t_i,a_i), the number t_i is called the *time stamp* of the action a_i, *i.e.* the time at which event a_i occurs.

Remark 3. Timed traces as defined above are also known as *strongly monotonic* timed traces, because of the assumption that no consecutive events occur at the same time. *Weakly* monotonic timed traces, *i.e.* with requirement $t_i \leq t_{i+1}$ instead of $t_i < t_{i+1}$, have also been considered, and there are some subtle differences between the two; see [94] for an important example.

Definition 19. *A timed trace* $((t_1,a_1),\dots,(t_k,a_k))$ *is* accepted *by a timed automaton* $A = (L,\ell_0,F,C,\Sigma,I,E)$ *if there is an accepting run*

$$(\ell_0,v_0) \xrightarrow{t_1} (\ell_0,v_0+t_1) \xrightarrow{a_1} (\ell_1,v_1) \xrightarrow{t_2-t_1} \cdots$$
$$\cdots \xrightarrow{a_{k-1}} (\ell_{k-1},v_{k-1}) \xrightarrow{t_k-t_{k-1}} (\ell_{k-1},v_{k-1}+t_k-t_{k-1}) \xrightarrow{a_k} (\ell_k,v_k)$$

in A*. The* timed language *of* A *is* $L(A) = \{\tau \in T\Sigma^* \mid \tau$ *accepted by* $A\}$*.*

It is clear that $L(A) = \emptyset$ if and only if none of the locations in F is reachable, hence Theorem 3 provides us with the decidability result in the following theorem. Undecidability of universality was established in [9]; we give an account of the proof below.

Theorem 7. *For a timed automaton $A = (L, \ell_0, F, C, \Sigma, I, E)$, deciding whether $L(A) = \emptyset$ is* PSPACE*-complete. It is undecidable whether $L(A) = T\Sigma^*$.*

Proof. We show that the universality problem for a timed automata is undecidable by reduction from the Σ_1^1-hard problem of deciding whether a given 2-counter machine M has a recurring computation.

Let the timed language L_u be the set of timed traces encoding recurring computations of M. Observe that $L_u = \emptyset$ if and only if M does not have such a computation. We then construct a timed automaton A_u which accepts the complement of L_u, *i.e.* $L(A_u) = T\Sigma^* \setminus L_u$. Hence the language of A_u is universal if and only if M does not have a recurring computation.

Recall that a 2-counter, or Minsky, machine M is a finite sequence of labeled instructions $\{I_0, \cdots, I_n\}$ and counters $\mathtt{x}_1$ and $\mathtt{x}_2$, with I_i for $0 \le i \le n-1$ on the form

$$I_i : \mathtt{x}_c \ \mathtt{:=}\ \mathtt{x}_c + \mathtt{1};\ \mathtt{goto}\ I_j \quad \textit{or} \quad I_i : \begin{cases} \mathtt{if}\ \mathtt{x}_c\ \mathtt{=}\ \mathtt{0}\ \mathtt{then}\ \mathtt{goto}\ I_j \\ \mathtt{else}\ \mathtt{x}_c\ \mathtt{=}\ \mathtt{x}_c\mathtt{-1};\ \mathtt{goto}\ I_k \end{cases}$$

for $c \in 1, 2$, with a special $I_n : \mathtt{Halt}$ instruction which stops the computation.

The language L_u is designed such that each I_i and the counters $\mathtt{x}_1$ and $\mathtt{x}_2$ are represented by actions in Σ. A correctly encoded computation is represented by a timed trace where “instruction actions” occur at discrete intervals, while the state (values of $\mathtt{x}_1$ and $\mathtt{x}_2$) is encoded by occurrences of “counter actions” in-between instruction actions (*e.g.* if $\mathtt{x}_i = 5$ after instruction I_j, then action x_i occurs 5 times within the succeeding interval of length 1).

When counters are incremented (or decremented), one more (or less) such action occurs through the next interval, and increments and decrements are always from the right. Additionally we require corresponding counter actions to occur exactly with a time difference of 1, such that if x_i occurs with time stamp a then also x_i occurs with time stamp $a+1$, unless x_i is the rightmost x_i action

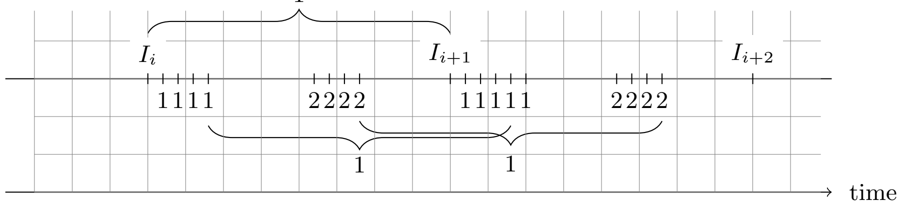


Fig. 5. Timed trace encoding a increment instruction I_{i+1} of a 2-counter machine

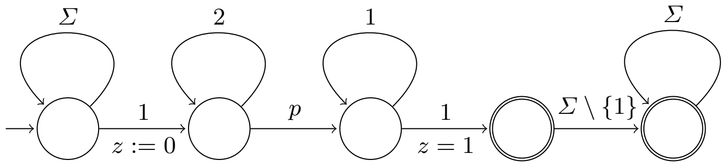


Fig. 6. Timed automaton which violates the encoding of the increment instruction

and I_i at time stamp $\lfloor a \rfloor$ is a decrement of $\mathtt{x}_i$. Figure 5 shows a increment of $\mathtt{x}_1$ (from 4 to 5) using actions 1 and 2.

We obtain A_u as a disjunction of timed automata $A^1, \dots, A^k$ where each A^i violates some property of a (correctly encoded) timed trace in L_u, either by accepting traces of incorrect format or inaccurate encodings of instructions.

Consider the instruction: `(p):` $\mathtt{x}_1$`:=` $\mathtt{x}_1$`+1 goto (q)`, incrementing $\mathtt{x}_1$ and jumping to `q`. A correct encoding would be similar to the one depicted in Figure 5 where all 1's and 2's are matched *one* time unit later, but with an additional 1 action occurring. In order to accept all traces except this encoding we must consider all possible violations, *i.e.*

– not incrementing the counter (no change),
– decrementing the counter,
– incrementing the counter more than once,
– jumping to the wrong instruction, or
– incrementing the wrong counter,

and construct a timed automaton having exactly such traces.

Figure 6 shows the timed automaton accepting traces in which instruction `p` yields no change of $\mathtt{x}_1$. □

Turning our attention to timed trace inclusion and equivalence, we note the following.

Proposition 5. *Let A and B be timed automata. If A is timed simulated by B, then $L(A) \subseteq L(B)$. If A and B are timed bisimilar, then $L(A) = L(B)$.*

By a standard argument, Theorem 7 implies undecidability of timed trace inclusion and equivalence, a result first shown in [8].

Theorem 8. *Timed trace inclusion and equivalence are undecidable for timed automata.*

There is also a notion of *untimed* traces for timed automata.

Definition 20. *The* untiming *of a set of timed traces $L \subseteq T\Sigma^*$ over a finite set of actions Σ is the set*

$$UL = \big\{ w = (a_1, \dots, a_k) \in \Sigma^* \;\big|\; \exists t_1, \dots, t_k \in \mathbb{R}_{\geq 0} : ((t_1, a_1), \dots, (t_k, a_k)) \in L \big\}.$$

Hence we have a notion of the set $UL(A)$ of *untimed language* of a timed automaton A. One can also define an untime operation U for timed automata, forgetting about the timing information of a timed automaton and thus converting it to a finite automaton; note however that $UL(A) \subsetneq L(UA)$ in general.

Lemma 1 ([9]). *For A a timed automaton, $UL(A) = L(\llbracket A \rrbracket_{\cong})$ provided that δ-transitions in $\llbracket A \rrbracket_{\cong}$ are taken as silent.*

As a corollary, sets of untimed traces accepted by timed automata are *regular*:

Theorem 9 ([9]). *For a timed automaton $A = (L, \ell_0, F, C, \Sigma, I, E)$, the set $UL(A) \subseteq \Sigma^*$ is regular. Accordingly, whether $UL(A) = \emptyset$ is decidable, and so is whether $UL(A) = \Sigma^*$. Also untimed trace inclusion and equivalence are decidable.*

2.6 Zones and Difference-Bound Matrices

As shown in the above sections, regions provide a finite and elegant abstraction of the infinite state space of timed automata, enabling us to prove decidability of reachability, timed and untimed bisimilarity, untimed language equivalence and language emptiness.

Unfortunately, the number of states obtained from the region partitioning is extremely large. In particular, by Proposition 3 the number of regions is exponential in the number of clocks as well as in the maximal constants of the timed automaton. Efforts have been made in developing more efficient representations of the state space [23, 28, 82], using the notion of *zones* from Definition 4 on page 70 as a coarser and more compact representation of the state space.

An extended clock constraint over a finite set C may be represented using a directed weighted graph, where the nodes correspond to the elements of C together with an extra "zero" node x_0, and an edge $x_i \xrightarrow{k} x_j$ corresponds to a constraint $x_i - x_j \le k$ (if there is more than one upper bound on $x_i - x_j$, k is the minimum of all these constraints' right-hand sides). The extra clock x_0 is fixed at value 0, so that a constraint $x_i \le k$ can be represented as $x_i - x_0 \le k$. Lower bounds on $x_i - x_j$ are represented as (possibly negative) upper bounds on $x_j - x_i$, and strict bounds $x_i - x_j < k$ are represented by adding a flag to the corresponding edge.

The weighted graph in turn may be represented by its adjacency matrix, which in this context is known as a *difference-bound matrix* or DBM. The above technique has been introduced in [55].

Example 3. Figure 7 gives an illustration of an extended clock constraint together with its representation as a difference-bound matrix. Note that the clock constraint contains superfluous information.

Zone-based reachability analysis of a timed automaton A uses symbolic states of the type (ℓ, Z), where ℓ is a location of A and Z is a zone, instead of the region-based symbolic states of Proposition 2.

$$Z = \begin{cases} x_1 \le 3 \\ x_1 - x_2 \le 10 \\ x_1 - x_2 \ge 4 \\ x_1 - x_3 \le 2 \\ x_3 - x_2 \le 2 \\ x_3 \ge -5 \end{cases}$$

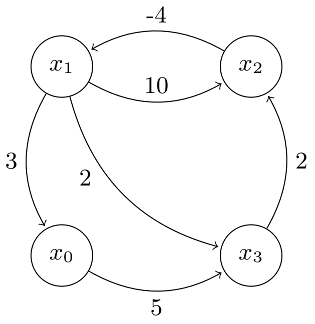


Fig. 7. Graph representation of extended clock constraint

Definition 21. *For a finite set C, $Z \subseteq \mathbb{R}_{\ge 0}^C$, and $r \subseteq C$, define*

- *the* delay *of Z by $Z^\uparrow = \{v + d \mid v \in Z, d \in \mathbb{R}_{\ge 0}\}$ and*
- *the* reset *of Z under r by $Z[r] = \{v[r] \mid v \in Z\}$.*

Lemma 2 ([69, 105]). *If Z is a zone over C and $r \subseteq C$, then $Z^\uparrow$ and $Z[r]$ are also zones over C.*

Extended clock constraints representing $Z^\uparrow$ and $Z[r]$ may be computed efficiently (in time cubic in the number of clocks in C) by representing the zone Z in a canonical form obtained by computing the *shortest-path closure* of the directed graph representation of Z, see [80].

Example 3 (continued). Figure 8 shows two canonical representations of the difference-bound matrix for the zone Z of Figure 7. The left part illustrates the shortest-path closure of Z; on the right is the *shortest-path reduction* [80] of Z, essentially obtained by removing redundant edges from the shortest-path closure. The latter is useful for checking zone inclusion, see below.

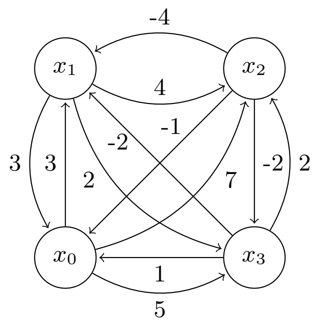
The *zone automaton* associated with a timed automaton is similar to the region automaton of Proposition 2, but uses zones for symbolic states instead of regions:

Definition 22. *The* zone automaton *associated with a timed automaton $A = (L, \ell_0, F, C, \Sigma, I, E)$ is the transition system $[\![A]\!]_Z = (S, s_0, \Sigma \cup \{\delta\}, T)$ given as follows:*

$$S = \big\{(\ell, Z) \bigm| \ell \in L, Z \subseteq \mathbb{R}_{\ge 0}^C \text{ zone}\big\} \qquad s_0 = \big(\ell_0, [\![v_0]\!]\big)$$
$$T = \big\{(\ell, Z) \overset{\delta}{\rightsquigarrow} \big(\ell, Z^\uparrow \wedge I(\ell)\big)\big\}$$
$$\cup \big\{(\ell, Z) \overset{a}{\rightsquigarrow} \big(\ell', (Z \wedge \varphi)[r] \wedge I(\ell')\big) \bigm| \ell \xrightarrow{\varphi, a, r} \ell' \in E\big\}$$

The analogue of Proposition 1 for zone automata is as follows:

Proposition 6 ([105]). *A state (ℓ, v) in a timed automaton $A = (L, \ell_0, F, C, \Sigma, I, E)$ is reachable if and only if there is a zone $Z \subseteq \mathbb{R}_{\ge 0}^C$ for which $v \in Z$ and such that (ℓ, Z) is reachable in $[\![A]\!]_Z$.*



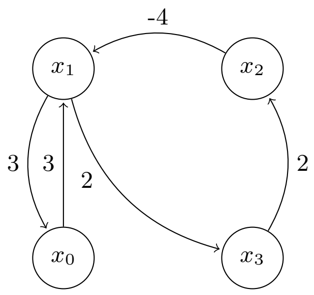


Fig. 8. Canonical representations. Left: shortest-path closure; right: shortest-path reduction

The zone automaton associated with a given timed automaton is *infinite* and hence unsuitable for reachability analysis. Finiteness can be enforced by employing *normalization*, using the fact that region equivalence $\cong$ has finitely many equivalence classes:

Definition 23. *For a timed automaton A and a zone $Z \subseteq \mathbb{R}_{\geq 0}^C$, the* normalization *of Z is the set* $\{v : C \to \mathbb{R}_{\geq 0} \mid \exists v' \in Z : v \cong v'\}$

The normalized zone automaton is defined in analogy to the zone automaton from above, and Proposition 6 also holds for the normalized zone automaton. Hence we can obtain a reachability algorithm by applying any search strategy (depth-first, breadth-first, or another) on the normalized zone automaton.

Remark 4. For timed automata on *extended* clock constraints, *i.e.* with diagonal constraints permitted, it can be shown [27, 32] that normalization as defined above does *not* give rise to a sound and complete characterization of reachability. Instead, one can apply a refined normalization which depends on the difference constraints used in the timed automaton, see [27].

In addition to the efficient computation of symbolic successor states according to the $\rightsquigarrow$ relation, termination of reachability analysis requires that we can efficiently recognize whether the search algorithm has encountered a given symbolic state. Here it is crucial that there is an efficient way of deciding inclusion $Z_1 \subseteq Z_2$ between zones. Both the shortest-path-closure canonical form as well as the more space-economical shortest-path-reduced canonical form [80], *cf.* Example 3, allow for efficient inclusion checking.

In analogy to difference-bound matrices and overcoming some of their problems, the data structure called *clock difference diagram* has been proposed [82]. However, the design of efficient algorithms for delay and reset operations over that data structure is a challenging open problem; generally, the design of efficient data structures for computations with (unions of) zones is a field of active research, see [3, 12, 71, 88, 93] for some examples.

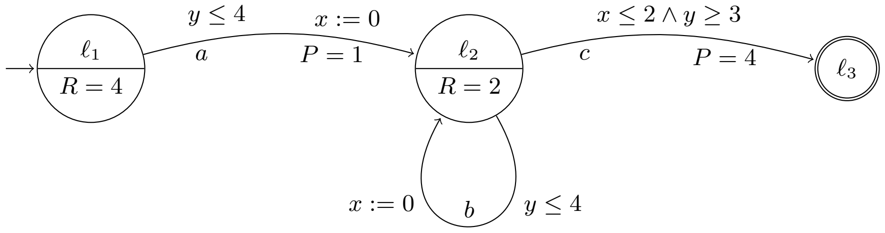


Fig. 9. A weighted timed automaton with two clocks

3 Weighted Timed Automata

The notion of *weighted* — or *priced* — timed automata was introduced independently, at the very same conference, by Behrmann *et.al.* [21] and Alur *et.al.* [10]. In these models both edges and locations can be decorated with weights, or prices, giving the cost of taking an action transition or the cost per time unit of delaying in a given location. The total cost of a trace is then simply the accumulated (or total) weight of its discrete and delay transitions.

As a first result, the above two papers independently, and with quite different methods, showed that the problem of cost-optimal reachability is computable for weighted timed automata with non-negative weights. Later, optimal reachability for timed automata with several weight functions was considered in [85] as well as optimal infinite runs in [34, 59].

Definition 24. *A* weighted timed automaton *is a tuple* $A = (L, \ell_0, F, C, \Sigma, I, E, R, P)$*, where* $(L, \ell_0, F, C, \Sigma, I, E)$ *is a timed automaton,* $R : L \to \mathbb{Z}$ *a location weight-rate mapping, and* $P : E \to \mathbb{Z}$ *an edge weight mapping.*

The semantics *of* A *is the weighted transition system* $[\![A]\!] = (S, s_0, \Sigma \cup \mathbb{R}_{\ge 0}, T, w)$*, where* $(S, s_0, \Sigma \cup \mathbb{R}_{\ge 0}, T)$ *is the semantics of the underlying timed automaton* $(L, \ell_0, F, C, \Sigma, I, E)$*, and the transition weights* $w : T \to \mathbb{R}$ *are given as follows:*

$$w\big((\ell, v) \xrightarrow{d} (\ell, v + d)\big) = d\, R(\ell)$$
$$w\big((\ell, v) \xrightarrow{a} (\ell', v')\big) = P\big(\ell \xrightarrow{\varphi, a, r} \ell'\big) \quad \textit{with } v \models \varphi, v' = v[r]$$

We shall denote weighted edges and transitions by symbols $\xrightarrow[w]{e}$ to illustrate an edge or a transition labeled e with weight w.

3.1 Optimal Reachability

The objective of optimal reachability analysis is to find runs to a final location with the lowest *total weight* as defined below.

Example 4. Figure 9 shows a simple weighted timed automaton with final location ℓ_3. Below we give a few examples of accepting runs, where we identify valuations $v : \{x, y\} \to \mathbb{R}_{\geq 0}$ with their values $(v(x), v(y))$. The total weights of the runs given here are 17 and 11; actually the second run is *optimal* in the sense of Problem 2 below:

$$(\ell_1, 0, 0) \xrightarrow[12]{3} (\ell_1, 3, 3) \xrightarrow[1]{a} (\ell_2, 0, 3) \xrightarrow[4]{c} (\ell_3, 0, 3)$$
$$(\ell_1, 0, 0) \xrightarrow[1]{a} (\ell_2, 0, 0) \xrightarrow[6]{3} (\ell_2, 3, 3) \xrightarrow[0]{b} (\ell_2, 0, 3) \xrightarrow[4]{c} (\ell_3, 0, 3)$$

Definition 25. *The* total weight *of a finite run* $\rho = s_0 \xrightarrow[w_1]{} s_1 \xrightarrow[w_2]{} \cdots \xrightarrow[w_k]{} s_k$ *in a weighted transition system is* $w(\rho) = \sum_{i=1}^{k} w_k$*.*

We are now in a position to state the problem with which we are concerned here: We want to find accepting runs with minimum total weight in a weighted timed automaton A. However, due to the possible use of strict clock constraints on edges and in locations of A, the minimum total weight might not be realizable, *i.e.* there might be no run which achieves it. For this reason, one also needs to consider (infinite) *sets* of runs and the infimum of their members' total weights:

Problem 2 (Optimal reachability). Given a weighted timed automaton A, compute $W = \inf \{ w(\rho) \mid \rho \text{ accepting run in } A \}$ and, for each $\varepsilon > 0$, an accepting run ρ_ε for which $w(\rho_\varepsilon) < W + \varepsilon$.

The key ingredient in the proof of the following theorem is the introduction of *weighted regions* in [21]. A weighted region is a region as of Definition 11 enriched with an affine cost function describing in a finite manner the cost of reaching any point within it. This notion allows one to define the weighted region automaton associated with a weighted timed automaton, and one can then show that optimal reachability can be computed in the weighted region automaton. PSPACE-hardness in the below theorem follows from PSPACE-hardness of reachability for timed automata.

Theorem 10 ([21]). *The optimal reachability problem for weighted timed automata with non-negative weights is* PSPACE*-complete.*

Similar to the notion of regions for timed automata, the number of weighted regions is exponential in the number of clocks as well as in the maximal constants of the timed automaton. Hence a notion of *weighted zone* — a zone extended with an affine cost function — was introduced [79] together with an efficient, symbolic A^*-algorithm for searching for cost-optimal tracing using branch-and-bound techniques. In particular, efficient means of generalizing the notion of symbolic successor to incorporate the affine cost functions were given.

During the symbolic exploration, several small linear-programming problems in terms of determining the minimal value of the cost function over the given zone have to be dealt with. Given that the constraints of these problems are simple difference constraints, it turns out that substantial gain in performance may be achieved by solving the dual problem of minimum-cost flow [98]. The Uppaal branch Uppaal-Cora provides an efficient tool for cost-optimal reachability analysis, applying the above data structures and algorithms and allowing the user to guide and heuristically prune the search.

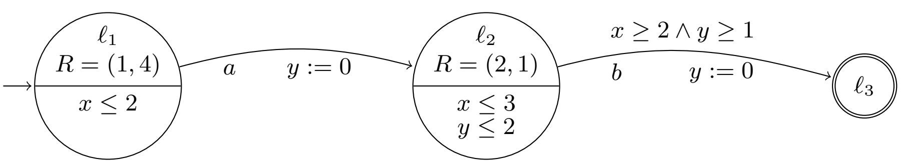


Fig. 10. A doubly weighted timed automaton with two clocks

3.2 Multi-weighted Timed Automata

The below formalism of doubly weighted timed automata is a generalization of weighted timed automata useful for modeling systems with several different resources.

Definition 26. *A* doubly weighted timed automaton *is a tuple*

$$A = (L, \ell_0, F, C, \Sigma, I, E, R, P)$$

where $(L, \ell_0, F, C, \Sigma, I, E)$ is a timed automaton, $R : L \to \mathbb{Z}^2$ a location weight-rate mapping, and $P : E \to \mathbb{Z}^2$ an edge weight mapping.

The semantics of a doubly weighted timed automaton is a doubly weighted transition system defined similarly to Definition 24, and the total weight of finite runs is defined accordingly as a pair; we shall refer to the total weights as w_1 and w_2 respectively. These definitions have natural generalizations to *multi-weighted* timed automata with more than two weight coordinates.

The objective of *conditional reachability* analysis is to find runs to a final location with the lowest total weight in the first weight coordinate while satisfying a constraint on the other weight coordinate.

Example 5. Figure 10 depicts a simple doubly weighted timed automaton with final location ℓ_3. Under the constraint $w_2 \leq 3$, the optimal run of the automaton can be seen to be

$$(\ell_1, 0, 0) \xrightarrow[(\frac{1}{3},\frac{4}{3})]{1/3} (\ell_1, 1/3, 1/3) \xrightarrow{a} (\ell_2, 1/3, 0) \xrightarrow[(\frac{10}{3},\frac{5}{3})]{5/3} (\ell_2, 2, 5/3) \xrightarrow{b} (\ell_3, 2, 0)$$

with total weight $\left(\frac{11}{3}, 3\right)$.

The precise formulation of the conditional optimal reachability problem is as follows, where we again need to refer to (possibly infinite) sets of runs:

Problem 3 (Conditional optimal reachability). Given a doubly weighted timed automaton A and $M \in \mathbb{Z}$, compute $W = \inf\{w_1(\rho) \mid \rho \text{ accepting run in } A, w_2(\rho) \leq M\}$ and, for each $\varepsilon > 0$, an accepting run ρ_ε for which $w_2(\rho) \leq M$ and $w_1(\rho_\varepsilon) < W + \varepsilon$.

Theorem 11 ([84, 85]). *The conditional optimal reachability problem is computable for doubly weighted timed automata with non-negative weights and without weights on edges.*

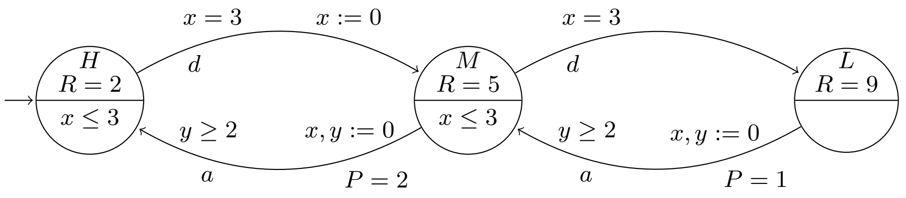


Fig. 11. A weighted timed automaton modelling a simple production system

The proof of the above theorem rests on a direct generalization of weighted to *doubly-weighted* zones. An extension can be found in [85], where it is shown that also the *Pareto frontier*, *i.e.* the set of cost vectors which cannot be improved in any cost variable, can be computed.

3.3 Optimal Infinite Runs

In this section we shall be concerned with computing optimal *infinite* runs in (doubly) weighted timed automata. We shall treat both the *limit ratio* viewpoint discussed in [34] and the *discounting* approach of [59, 60].

Example 6. Figure 11 shows a simple production system modelled as a weighted timed automaton. The system has three modes of production, High, Medium, and Low. The weights model the *cost* of production, so that the High production mode has a low cost, which is preferable to the high cost of the Low production mode. After operating in a High or Medium production mode for three time units, production automatically degrades (action d) to a lower mode. When in Medium or Low production mode, the system can be attended to (action a), which advances it to a higher mode.

The objective of *optimal-ratio analysis* is to find an infinite run in a doubly weighted timed automaton which minimizes the *ratio* between the two total weights. This will be formalized below.

Definition 27. *The* total ratio *of a finite run* $\rho = s_0 \xrightarrow[z_1]{w_1} s_1 \xrightarrow[z_2]{w_2} \cdots \xrightarrow[z_k]{w_k} s_k$ *in a doubly weighted transition system is*

$$\Gamma(\rho) = \frac{\sum_{i=1}^{k} w_k}{\sum_{i=1}^{k} z_k}.$$

The total ratio of an infinite run $\rho = s_0 \xrightarrow[z_1]{w_1} s_1 \xrightarrow[z_2]{w_2} \cdots$ *is*

$$\Gamma(\rho) = \liminf_{k\to\infty} \Gamma(s_0 \to \cdots \to s_k).$$

A special case of optimal-ratio analysis is given by weight-per-time models, where the interest is in minimizing total weight per accumulated time. The example provided in this section is a case of this. In the setting of optimal-ratio analysis, these can be modelled as doubly weighted timed automata with $R_2(\ell) = 1$ and $P_2(e) = 0$ for all locations ℓ and edges e.

Example 6 (continued). In the timed automaton of Figure 11, the following cyclic behaviour provides an infinite run ρ:

$$(H,0,0) \xrightarrow{3} (H,3,3) \xrightarrow{d} (M,0,3) \xrightarrow{3} (M,3,6) \xrightarrow{d} (L,3,6) \xrightarrow{1} \\ (L,4,7) \xrightarrow{a} (M,0,0) \xrightarrow{3} (M,3,3) \xrightarrow{a} (H,0,0) \to \cdots$$

Taking the weight-per-time viewpoint, the total ratio of ρ is $\Gamma(\rho) = 4.8$.

Problem 4 (Minimum infinite ratio). Given a doubly weighted timed automaton A, compute $W = \inf \{\Gamma(\rho) \mid \rho \text{ infinite run in } A\}$ and, for each $\varepsilon > 0$, an infinite run ρ_ε for which $\Gamma(\rho_\varepsilon) < W + \varepsilon$.

The main tool in the proof of the following theorem is the introduction of the *corner-point abstraction* of a timed automaton in [34]. This is a finite refinement of the region automaton of Definition 11 in which one also keeps track of the corner points of regions. One can then show that any infinite run with minimum ratio must pass through corner points of regions, hence these can be found in the corner-point abstraction by an algorithm first proposed in [76].

The technical condition in the theorem that the second weight coordinate be *strongly diverging* means that any infinite run ρ in the closure of the timed automaton in question satisfies $w_2(\rho) = \infty$, see [34] for details.

Theorem 12 ([34]). *The minimum infinite ratio problem is computable for doubly weighted timed automata with non-negative and strongly diverging second weight coordinate.*

For *discount-optimal analysis*, the objective is to find an infinite run in a weighted timed automaton which minimizes the *discounted total weight* as defined below. The point of discounting is that the weight of actions is discounted with time, so that the impact of an event decreases, the further in the future it takes place.

In the definition below, ε is the empty run, and $(\ell, v) \to \rho$ denotes the concatenation of the transition $(\ell, v) \to$ with the run ρ.

Definition 28. *The* discounted total weight *of finite runs in a weighted timed automaton under discounting factor $\lambda \in [0, 1[$ is given inductively as follows:*

$$\begin{aligned} w_\lambda(\varepsilon) &= 0 \\ w_\lambda\big((\ell, v) \xrightarrow[P]{a} \rho\big) &= P + w_\lambda(\rho) \\ w_\lambda\big((\ell, v) \xrightarrow{d} \rho\big) &= R(\ell) \int_0^d \lambda^\tau d\tau + \lambda^d w_\lambda(\rho) \end{aligned}$$

The discounted total weight of an infinite run $\rho = (\ell_0, v_0) \xrightarrow{d_1} (\ell_0, v_0 + d_1) \xrightarrow[P_1]{a_1} (\ell_1, v_1) \to \cdots$ *is*

$$w_\lambda(\rho) = \lim_{k\to\infty} w_\lambda\big((\ell_0, v_0) \to \cdots \xrightarrow[P_k]{a_k} (\ell_k, v_k)\big)$$

provided that the limit exists.

Example 6 (continued). The discounted total weight of the infinite run ρ in the timed automaton of Figure 11 satisfies the following equality, where $I_t = \int_0^t \lambda^\tau d\tau = -\frac{1}{\ln\lambda}(1 - \lambda^t)$:

$$w_\lambda(\rho) = 2I_3 + \lambda^3(5I_3 + \lambda^3(9I_1 + \lambda(1 + 5I_3 + \lambda^3(2 + w_\lambda(\rho)))))$$

With a discounting factor of $\lambda = .9$ for example, the discounted total weight of ρ would hence be $w_\lambda(\rho) \approx 40.5$.

Problem 5 (Minimum discounted weight). Given a weighted timed automaton A and $\lambda \in [0, 1[$, compute $W = \inf\big\{w_\lambda(\rho) \bigm| \rho \text{ infinite run in } A\big\}$ and a set P of infinite runs for which $\inf_{\rho\in P} w_\lambda(\rho) = W$.

The proof of the following theorem rests again on the corner-point abstraction, and on a result in [11]. The technical condition that the timed automaton be time-divergent is analogous to the condition on the second weight coordinate in Theorem 12.

Theorem 13 ([59]). *The minimum discounted weight problem is computable for time-divergent weighted timed automata with non-negative weights and rational* λ.

3.4 Energy Problems

Instead of considering various forms of (conditional) optimality, one can also concern oneself simply with the question whether there *exists* an infinite run satisfying certain bounds on the weights of finite prefixes. This so-called *energy problem*, which is surprisingly intricate, was first introduced in [37] and has since been dealt with *e.g.* in [36,38,47,48,58,97].

For an infinite run $\rho = s_0 \xrightarrow[w_1]{} s_1 \xrightarrow[w_2]{} \cdots$ and a positive integer k, we write $\rho_{\rceil k}$ for the finite prefix $\rho_{\rceil k} = s_0 \xrightarrow[w_1]{} \cdots \xrightarrow[w_k]{} s_k$ of ρ.

Problem 6 (Energy problems). Given a weighted timed automaton A and a positive integer M, decide whether there exists an infinite run ρ in A for which $w(\rho_{\rceil k}) \geq 0$ for all $k \in \mathbb{N}$, and whether there exists an infinite run σ in A for which $0 \leq w(\sigma_{\rceil k}) \leq M$ for all $k \in \mathbb{N}$.

The first of these problems is called the *lower bound* energy problem, the second one the *interval bound* energy problem. There is a third variant of the problem which we do not treat here; see [37] for details.

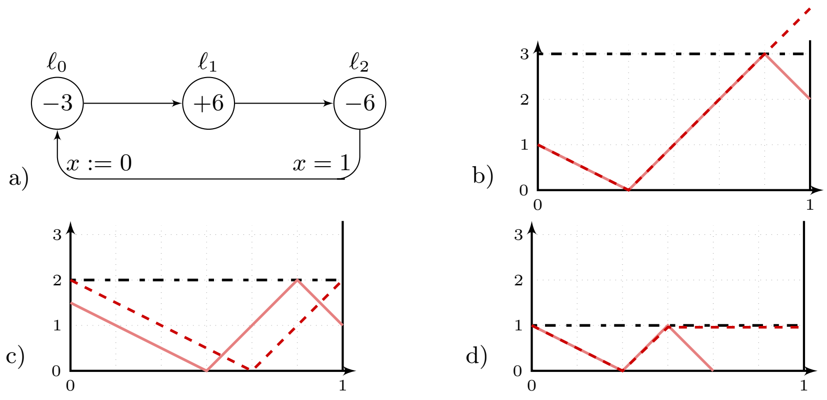


Fig. 12. a) Weighted timed automaton (global clock invariant $x \leq 1$) b) $U = +\infty$ and $U = 3$. c) $U = 2$. d) $U = 1$ and $W = 1$.

Example 7. Consider the weighted timed automaton in Figure 12a) with infinite behaviours repeatedly delaying in ℓ_0, ℓ_1 and ℓ_2 for a total of precisely one time-unit. Let us observe the effect of lower and upper constraints on the energy-level on so-called bang-bang strategies, where the behaviour remains in a given location as long as permitted by the given bounds. Figure 12b) shows the bang-bang strategy given an initial energy-level of 1 with no upper bound (dashed line) or 3 as upper bound (solid line). In both cases, it may be seen that the bang-bang strategy yields an infinite run.

In Figures 12c) and d), we consider the upper bounds 2 and 1, respectively. For an upper bound of 2, we see that the bang-bang strategy reduces an initial energy-level of $1\frac{1}{2}$ to 1 (solid line), and yet another iteration will reduce the remaining energy-level to 0. In fact, the bang-bang strategy—and it may be argued, any other strategy—fails to maintain an infinite behaviour for any initial energy-level *except for* 2 (dashed line). With upper-bound 1, the bang-bang strategy—and any other strategy—fails to complete even one iteration (solid line).

4 Timed Games

Recently, substantial effort has been made towards the synthesis of winning strategies for timed games with respect to *safety* and *reachability control objectives*. From known region-based decidability results, efficient on-the-fly algorithms have been developed [44, 102] and implemented in the newest branch Uppaal-Tiga.

For timed games, as for untimed ones, transitions are either controllable or uncontrollable (*i.e.* under the control of an environment), and the problem is to synthesize a strategy for *when* to take *which* (enabled) controllable transitions in order that a given objective is guaranteed regardless of the behaviour of the environment.

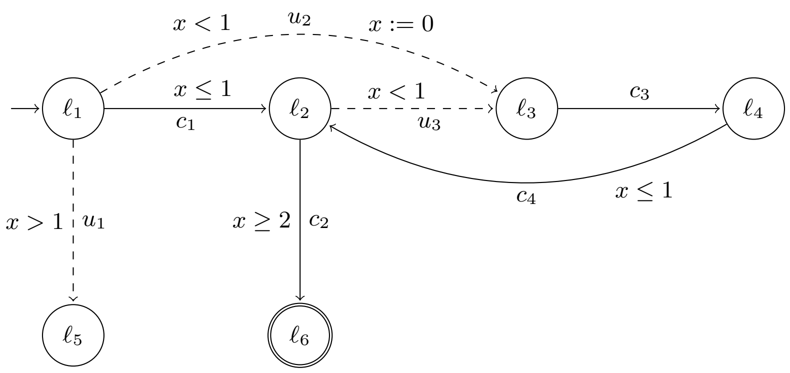


Fig. 13. A timed game with one clock. Controllable edges (with actions from Σ_c) are solid, uncontrollable edges (with actions from Σ_u) are dashed.

Definition 29. *A* timed game *is a tuple* $(L, \ell_0, F, C, \Sigma_c, \Sigma_u, I, E)$ *with* $\Sigma_c \cap \Sigma_u = \emptyset$ *and for which the tuple* $(L, \ell_0, F, C, \Sigma = \Sigma_c \cup \Sigma_u, I, E)$ *is a timed automaton.*

Edges with actions in Σ_c are said to be *controllable*, those with actions in Σ_u are *uncontrollable.*

Example 8. Figure 13 provides a simple example of a timed game. Here, $\Sigma_c = \{c_1, c_2, c_4\}$ and $\Sigma_2 = \{u_1, u_2, u_3\}$, and the controllable edges are drawn with solid lines, the uncontrollable ones with dashed lines.

We need the notion of *strategy*; essentially, a strategy provides instructions for which controllable edge to take, or whether to wait, in a given state:

Definition 30. *A* strategy *for a timed game* $A = (L, \ell_0, F, C, \Sigma_c, \Sigma_u, I, E)$ *is a mapping* σ *from finite runs of* A *to* $\Sigma_c \cup \{\delta\}$, *where* $\delta \notin \Sigma$, *such that for any run* $\rho = (\ell_0, v_0) \to \cdots \to (\ell_k, v_k)$,

- *if* $\sigma(\rho) = \delta$, *then* $(\ell, v) \xrightarrow{d} (\ell, v + d)$ *in* $[\![A]\!]$ *for some* $d > 0$, *and*
- *if* $\sigma(\rho) = a$, *then* $(\ell, v) \xrightarrow{a} (\ell', v')$ *in* $[\![A]\!]$.

A strategy σ *is said to be* memoryless *if* $\sigma(\rho)$ *only depends on the last state of* ρ, *i.e. if* $\rho_1 = (\ell_0, v_0) \xrightarrow{d_1} (\ell_0, v_0 + d_1) \to \cdots \to (\ell_k, v_k)$, $\rho_2 = (\ell_0, v_0) \xrightarrow{d_1'} (\ell_0, v_0 + d_1') \to \cdots \to (\ell_k, v_k)$ *imply* $\sigma(\rho_1) = \sigma(\rho_2)$.

An *outcome* of a strategy is any run which adheres to its instructions in the obvious manner:

Definition 31. *A run* $(\ell_0, v_0) \xrightarrow{d_1} (\ell_0, v_0 + d_1) \to \cdots \to (\ell_k, v_k)$ *in a timed game* $A = (L, \ell_0, F, C, \Sigma_c, \Sigma_u, I, E)$ *is said to be an* outcome *of a strategy* σ *provided that*

– *for all* $(\ell_i, v_i) \xrightarrow{d} (\ell_i, v_i + d)$ *and for all* $d' < d$, *we have* $\sigma\big((\ell_0, v_0) \to \cdots \to (\ell_i, v_i + d')\big) = \delta$, *and*
– *for all* $(\ell_i, v_i + d) \xrightarrow{a} (\ell_{i+1}, v_{i+1})$ *for which* $a \in \Sigma_c$, *we have* $\sigma\big((\ell_0, v_0) \to \cdots \to (\ell_i, v_i')\big) = a$.

An outcome is said to be maximal *if* $\ell_k \in F$, *or if* $(\ell_k, v_k) \xrightarrow{a} (\ell_{k+1}, v_{k+1})$ *implies* $a \in \Sigma_u$.

Hence an outcome is maximal if it stops in a final state, or if no controllable actions are available at its end. An underlying assumption is that uncontrollable actions cannot be forced, hence a maximal outcome which does not end in a final state may "get stuck" in a non-final state. The aim of reachability games is to find strategies all of whose maximal outcomes end in a final state; the aim of safety games is to find strategies all of whose (not necessarily maximal) outcomes avoid final states:

Definition 32. *A strategy is said to be* winning for the reachability game *if any of its maximal outcomes is an accepting run. It is said to be* winning for the safety game *if none of its outcomes are accepting.*

Example 8 (continued). The following memoryless strategy is winning for the reachability game on the timed game from Figure 13:

$$\sigma(\ell_1, v) = \begin{cases} \delta & \text{if } v(x) \neq 1 \\ c_1 & \text{if } v(x) = 1 \end{cases} \qquad \sigma(\ell_2, v) = \begin{cases} \delta & \text{if } v(x) < 2 \\ c_2 & \text{if } v(x) \geq 2 \end{cases}$$

$$\sigma(\ell_3, v) = \begin{cases} \delta & \text{if } v(x) < 1 \\ c_3 & \text{if } v(x) \geq 1 \end{cases} \qquad \sigma(\ell_4, v) = \begin{cases} \delta & \text{if } v(x) \neq 1 \\ c_4 & \text{if } x(x) = 1 \end{cases}$$

Problem 7 (Reachability and safety games). Given a timed game A, does there exist a winning strategy for the reachability game on A? Does there exist a winning strategy for the safety game on A?

An important ingredient in the proof of the following theorem is the fact that for reachability as well as safety games, it is sufficient to consider *memoryless* strategies. This is not the case for other, more subtle, control objectives (*e.g.* counting properties modulo some N) as well as for the synthesis of winning strategies under *partial observability.*

Theorem 14 ([13, 91]). *The reachability and safety games are decidable for timed games.*

In [45] the on-the-fly algorithm applied in Uppaal-Tiga has been extended to timed games under partial observability.

The field of timed games is a very active research area. Research has been conducted towards the synthesis of *optimal* winning strategies for reachability games on *weighted timed games.* In [6, 35] computability of optimal strategies is shown under a certain condition of *strong cost non-zenoness*, requiring that the total weight diverges with a given minimum rate per time. Later undecidability results [33,41] show that for weighted timed games with three or more clocks this condition (or a similar one) is necessary. Lately [39] proves that optimal reachability strategies are computable for one-clock weighted timed games, though there is an unsettled (large) gap between the known lower bound complexity P and an upper bound of 3EXPTIME.

We conclude this section by reestablishing the connection between the notion of games and bisimulation [100] in the presence of time:

Proposition 7. *There is a polynomial time reduction from timed bisimilarity to timed safety games.*

Observe that this provides an alternative proof of the decidability of timed bisimilarity in Theorem 5 on page 76.

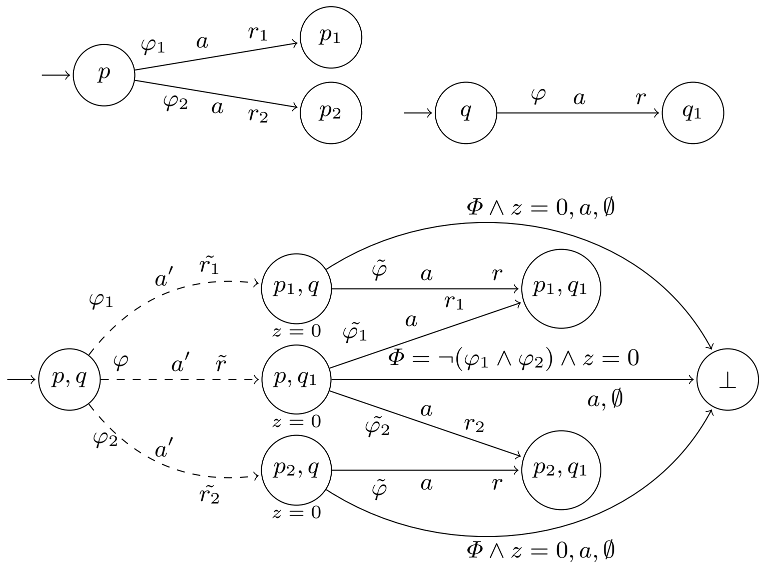
Proof. Given timed automata $A_i = (L_i, \ell_0^i, F, C_i, \Sigma, I_i, E_i)$, for $i \in \{1,2\}$, with $C_1 \cap C_2 = \emptyset$, we consider the timed game with locations $L = \{\bot\} \cup (L_1 \times L_2) \cup (L_1 \times L_2 \times \Sigma \times \{1,2\})$, where $F = \{\bot\}$ is a designated final location. We set $C = C_1 \cup C_2 \cup \{z\}$, where $z \notin C_1 \cup C_2$ is a fresh clock, $\Sigma_c = \Sigma \cup \{\bot\}$ and $\Sigma_u = \{a' \mid a \in \Sigma\}$, and E is defined by

$$\begin{aligned}
&(p,q) \xrightarrow{\varphi,a',\tilde r} (p',q,a)_1 \in E \quad \text{if} \quad p \xrightarrow{\varphi,a,r} p' \in E_1,\\
&(p,q) \xrightarrow{\varphi,a',\tilde r} (p,q',a)_2 \in E \quad \text{if} \quad q \xrightarrow{\varphi,a,r} q' \in E_2,\\
&(p',q,a)_1 \xrightarrow{\tilde\varphi,a,r} (p',q') \in E \quad \text{if} \quad q \xrightarrow{\varphi,a,r} q' \in E_2,\\
&(p,q',a)_2 \xrightarrow{\tilde\varphi,a,r} (p',q') \in E \quad \text{if} \quad p \xrightarrow{\varphi,a,r} p' \in E_1, \text{ and}\\
&(p,q,a)_i \xrightarrow{\Phi\wedge z=0,\bot,\emptyset} \bot \quad \text{for all } i \in \{1,2\}.
\end{aligned}$$

Here we denote $\tilde r = r \cup \{z\}$ and $\tilde\varphi = \varphi \wedge z = 0$, and $\Phi = \bigvee_j \neg\varphi_j$ when $q \xrightarrow{\varphi_j,a,r} q'$ and $i = 1$ and symmetrically for $i = 2$. Location invariants are defined by $I(p,q) = I_1(p) \wedge I_2(q)$ for $(p,q) \in L_1 \times L_2$ and $I(p,q,a)_i = (z = 0)$ for all $(p,q,a)_i \in L_1 \times L_2 \times \Sigma \times \{1,2\}$. See Figure 14 for a simple example of this construction.

It remains to be seen that A_1 and A_2 are timed bisimilar if and only if a strategy σ exists for which any outcome $\rho = (\ell_0, v_0) \xrightarrow{d_1} (\ell_0, v_0 + d_1) \to \cdots \to (\ell_k, v_k)$ satisfies $\ell_k \neq \bot$ (*i.e.* it avoids the final location $\bot$).

Assume A_1 and A_2 are timed bisimilar, then we can prove something stronger than the above, namely that *any* strategy will avoid $\bot$. Indeed, if $(\ell_0, v_0) \xrightarrow{d_1} (\ell_0, v_0 + d_1) \to \cdots \to (\ell_k, v_k)$ is a run in the timed game, and the transition $(\ell_{k-2}, v_{k-2}) \xrightarrow{a} (\ell_{k-1}, v_{k-1})$ exists due to the first component (p_{k-2}, v_{k-2}) of (ℓ_{k-2}, v_{k-2}), then the corresponding $\xrightarrow{a}$ transition in $[\![A_1]\!]$ has a matching $\xrightarrow{a}$ transition in $[\![A_2]\!]$. Hence the state (ℓ_{k-1}, v_{k-1}) has an enabled a-labeled edge, which by definition of Φ implies that the edge to $\bot$ is disabled, thus $\ell_k \neq \bot$. A symmetric argument applies in the other case.



φ_1 a r_1
φ_2 a r_2
p
p_1
p_2
q
φ a r
q_1
$\Phi \wedge z = 0, a, \emptyset$
p, q
p_1, q
p, q_1
p_2, q
p_1, q_1
p_2, q_1
$\perp$
$z = 0$
φ_1 a' $\tilde{r_1}$
φ a' $\tilde{r}$
φ_2 a' $\tilde{r_2}$
$\tilde{\varphi}$ a r
$\tilde{\varphi_1}$ a r_1
$\tilde{\varphi_2}$ a r_2
$\Phi = \neg(\varphi_1 \wedge \varphi_2) \wedge z = 0$
$a, \emptyset$

Fig. 14. A timed game constructed for bisimilarity checking of two simple timed automata

Now assume σ is a strategy which ensures avoidance of $\perp$, then we shall show that any $(\ell_j, v) = ((p_j, q_j), v)$ for $j \geq 0$, (*i.e.* of type $S_1 \times S_2$) occurring in an outcome of σ satisfies $(p_j, v) \sim (q_j, v)$. Assume to the contrary that $(p_j, v) \xrightarrow{a} (p'_j, v') \in [\![A_1]\!]$ and $(q_j, v) \overset{a}{\not\to} (q'_j, v'') \in [\![A_2]\!]$ for some $a \in \Sigma$, then we may extend the run to (ℓ_j, v) by $(\ell_j, v) \xrightarrow{a} (\ell_{j+1}, v') \xrightarrow{\perp} \perp$, moreover $\sigma((\ell_0, v_0) \to \cdots \to (\ell_{j+1}, v')) = \perp$ is the only choice for σ as by construction neither δ (due to the invariant $z = 0$) nor any $b \neq a$ is available. $\square$

5 Statistical Model Checking for Networks of Price Timed Automata

A weak point of model checking is undoubtly the state-space explosion, i.e. the exponential growth in the analysis effort measured in the number of model-components. Another limitation of real-time model checking is that it merely provides – admittedly most important – hard quantitative guarantees, e.g. the worst case response time of a recurrent task under a certain scheduling principle, the worst case execution time of a piece of code running on a particular execution platform, or the worst case time before consensus is reached by a real-time network protocol. In addition to these hard guarantees, it would be desirable in several situations to obtain refined performance information concerning likely or expected behaviors in terms of timing and resource consumption. In particular, this would allow to distinguish and select between systems that perform identically from a worst-case perspective.

In a series of recent works [53], we proposed a stochastic semantics for Priced Timed Automata (PTA), whose clocks can evolve with different rates, while[1] being used with no restrictions in guards and invariants. Networks of PTAs (NPTA) are created by composing PTAs via input and output actions. More precisely, we define a natural stochastic semantics for networks of NPTAs based on races between components being composed. We shall observe that such race can generate arbitrarily complex stochastic behaviors from simple assumptions on individual components. We shall see that our semantics cannot be emulated by applying the existing stochastic semantic of [14,30] to the product of components. Other related work includes the very rich framework of stochastic timed systems of MoDeST [31]. Here, however, general hybrid variables are not considered and parallel composition does not yield fully stochastic models. For the notion of probabilistic hybrid systems considered in [101] the choice of time is resolved non-deterministically rather than stochastically as in our case. Moreover, based on the stochastic semantics, we are able to express refined performance properties, e.g. in terms of probabilistic guarantees of time- and cost-bounded properties[2].

To allow for the efficient analysis of probabilistic performance properties we propose to work with Statistical Model Checking (SMC) [99, 106], an approach that has been proposed as an alternative to avoid an exhaustive exploration of the state-space of the model. The core idea of SMC is to monitor some simulations of the system, and then use results from the statistic area (including sequential hypothesis testing or Monte Carlo simulation) in order to decide whether the system satisfies the property with some degree of confidence.

In this section, we first give insights on the model and on the stochastic semantic, then on the use of statistical model checking. Finally, we conclude with a brief discussion and some applications;

5.1 Networks of Stochastic Automata

Networks of Price Timed Automata We consider the analysis of Priced Timed Automata (PTAs) that are timed automata whose clocks can evolve with different rates in different locations. In fact, the expressive power (up to timed bisimilarity) of NPTA equals that of general linear hybrid automata (LHA) [4], rendering most problems – including that of reachability – undecidable. We also assume PTAs are input-enabled, deterministic (with a probability measure defined on the sets of successors), and non-zeno. PTAs communicate via broadcast channels and shared variables to generate Networks of Price Timed Automata (NPTA).

Fig. 15 provides an NPTA with three components A, B, and T as specified using the UPPAAL GUI. One can easily see that the composite system $(A|B|T)$ has the transition sequence:

$((A_0, B_o, T_0), [x = 0, y = 0, C = 0]) \xrightarrow{1}\xrightarrow{a!}$
$((A_1, B_0, T_1), [x = 1, y = 1, C = 4]) \xrightarrow{1}\xrightarrow{b!}$

[1] In contrast to the usual restriction of priced timed automata [10].
[2] Clocks with different rates can be used to model costs.

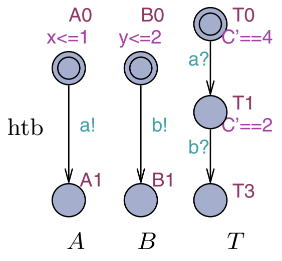


Fig. 15. An NPTA, $(A|B|T)$

$\big((A_1, B_1, T_2), [x = 2, y = 2, C = 6]\big)$,

demonstrating that the final location T_3 of T is reachable. In fact, location T_3 is reachable within cost 0 to 6 and within total time 0 and 2 in $(A|B|T)$ depending on when (and in which order) A and B choose to perform the output actions $a!$ and $b!$. Assuming that the choice of these time-delays is governed by probability distributions, a measure on sets of runs of NPTAs is induced, according to which quantitative properties such as *"the probability of T_3 being reached within a total cost-bound of 4.3"* become well-defined.

Probabilistic Semantics of NPTA Components. In our early works [53], we provide a natural stochastic semantics, where PTA components associate probability distributions to both the time-delays spent in a given state as well as to the transition between states. In Uppaal-SMC uniform distributions are applied for bounded delays and exponential distributions for the case where a component can remain indefinitely in a state. In a network of PTAs the components repeatedly race against each other, i.e. they independently and stochastically decide on their own how much to delay before outputting, with the "winner" being the component that chooses the minimum delay. For instance, in the NPTA of Fig. 15, A wins the initial race over B with probability 0.75.

In contrast to the probabilistic semantics of timed automata in [14, 30] our semantics deals with networks and thus with races between components. Let $\mathcal{A}^j = (L^j, X^j, \Sigma, E^j, R^j, I^j)$ $(j = 1 \ldots n)$ be a collection of composable NPTAs. Under the assumption of input-enabledness, disjointness of clock sets and output actions, states of the the composite NPTA $\boldsymbol{\mathcal{A}} = (\mathcal{A}_1 \,|\, \ldots \,|\, \mathcal{A}_n)$ may be seen as tuples $\mathbf{s} = (s_1, \ldots, s_n)$ where s_j is a state of $\mathcal{A}^j$, i.e. of the form (ℓ, ν) where $\ell \in L^j$ and $\nu \in \mathbb{R}_{\geq 0}^{X^j}$. Our probabilistic semantics is based on the principle of independency between components. Repeatedly each component decides on its own – based on a given delay density function and output probability function – how much to delay before outputting and what output to broadcast at that moment. Obviously, in such a race between components the outcome will be determined by the component that has chosen to output after the minimum delay: the output is broadcast and all other components may consequently change state.

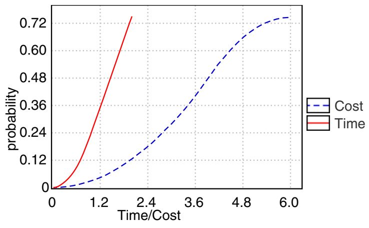


Fig. 16. Cumulative probabilities for `time` and `Cost`-bounded reachability of T_3

Let us first consider a component $\mathcal{A}^j$ and let St^j denote the corresponding set of states. For each state $s = (\ell, \nu)$ of $\mathcal{A}^j$ we shall provide probability distributions for both delays and outputs. In this presentation, we restrict to uniform and universal distributions, but arbitrary distributions can be considered.

The *delay density function* μ_s over delays in $\mathbb{R}_{\geq 0}$ will be either a uniform or an exponential distribution depending on the invariant of ℓ. Denote by E_ℓ the disjunction of guards g such that $(\ell, g, o, -, -) \in E^j$ for some output o. Denote by $d(\ell, \nu)$ the infimum delay before enabling an output, i.e. $d(\ell, \nu) = \inf\{d \in \mathbb{R}_{\geq 0} : \nu + R^j \cdot d \models E_\ell\}$, and denote by $D(\ell, \nu)$ the supremum delay, i.e. $D(\ell, \nu) = \sup\{d \in \mathbb{R}_{\geq 0} : \nu + R^j \cdot d \models I^j(\ell)\}$. If $D(\ell, \nu) < \infty$ then the delay density function μ_s is a uniform distribution on $[d(\ell, \nu), D(\ell, \nu)]$. Otherwise – that is $I^j(\ell)$ does not put an upper bound on the possible delays out of s – the delay density function μ_s is an exponential distribution with a rate $P(\ell)$, where $P : L^j \to \mathbb{R}_{\geq 0}$ is an *additional* distribution rate component added to the NPTA $\mathcal{A}^j$. For every state $s = (\ell, \nu)$, the *output probability function* γ_s over Σ_o^j is the uniform distribution over the set $\{o : (\ell, g, o, -, -) \in E^j \wedge \nu \models g\}$ whenever this set is non-empty[3]. We denote by s^o the state after the output of o. Similarly, for every state s and any input action ι, we denote by s^ι the state after having received the input ι.

Probabilistic Semantics of Networks of NPTA. We shall now see that while the stochastic semantics of each PTA is rather simple (but quite realistic), arbitrarily complex stochastic behavior can be obtained by their composition.

Reconsider the closed network $\boldsymbol{\mathcal{A}} = (\mathcal{A}_1 \,|\, \ldots \,|\, \mathcal{A}_n)$ with a state space $St = St_1 \times \cdots \times St_n$. For $\mathbf{s} = (s_1, \ldots, s_n) \in St$ and $a_1 a_2 \ldots a_k \in \Sigma^*$ we denote by $\pi(\mathbf{s}, a_1 a_2 \ldots a_k)$ the set of all maximal runs from $\mathbf{s}$ with a prefix $t_1 a_1 t_2 a_2 \ldots t_k a_k$ for some $t_1, \ldots, t_n \in \mathbb{R}_{\geq 0}$, that is runs where the i'th action a_i has been outputted by the component $A_{c(a_i)}$. We now inductively define the following measure for such sets of runs:

$$\mathsf{P}_{\mathsf{A}}\big(\pi(\mathbf{s}, a_1 \ldots a_n)\big) = \int_{t \geq 0} \mu_{s_c}(t) \cdot \Big(\prod_{j \neq c} \int_{\tau > t} \mu_{s_j}(\tau) d\tau\Big) \cdot \gamma_{s_c^t}(a_1) \cdot \mathsf{P}_{\mathsf{A}}\big(\pi(\mathbf{s}^t)^{a_1}, a_2 \ldots a_n)\big) dt$$

where $c = c(a_1)$, and as base case we take $P_{\mathsf{A}}(\pi(\mathbf{s}), \varepsilon) = 1$.

[3] Otherwise a specific weight distribution can be specified and used instead.

This definition requires a few words of explanation: at the outermost level we integrate over all possible initial delays t. For a given delay t, the outputting component $c = c(a_1)$ will choose to make the broadcast at time t with the stated density. Independently, the other components will choose to a delay amount, which – in order for c to be the winner – must be larger than t; hence the product of the probabilities that they each make such a choice. Having decided for making the broadcast at time t, the probability of actually outputting a_1 is included. Finally, in the global state resulting from all components having delayed t time-units and changed state according to the broadcasted action a_1 the probability of runs according to the remaining actions $a_2 \dots a_n$ is taken into account.

The Hammer Game. To illustrate the stochastic semantics further consider the network of two priced timed automata in Fig. 17 modeling a competition between the two players Axel and Alex both having to hammer three nails down. As can be seen by the representing `Work`-locations the time (-interval) and rate of energy-consumption required for hammering a nail depends on the player and the nail-number. As expected Axel is initially quite fast and uses a lot of energy but becomes slow towards the last nail, somewhat in contrast to Alex. To make it an interesting competition, there is only *one* hammer illustrated by repeated competitions between the two players in the `Ready`-locations, where the slowest player has to wait in the `Idle`-location until the faster player has finished hammering the next nail. Interestingly, despite the somewhat different strategy applied, the best- and worst-case completion times are identical for Axel and Alex: 59 seconds and 150 seconds. So, there is no difference between the two players and their strategy, or is there?

Assume now that a third person wants to bet on who is the more likely winner – Axel or Alex – given a refined semantics, where the time-delay before performing an output is chosen stochastically (e.g. by drawing from a uniform distribution) and independently by each player (component).

Under such a refined semantics there is a significant difference between the two players (Axel and Alex) in the Hammer Game. In Fig. 18a) the probability distributions for either of the two players winning before a certain time is given.

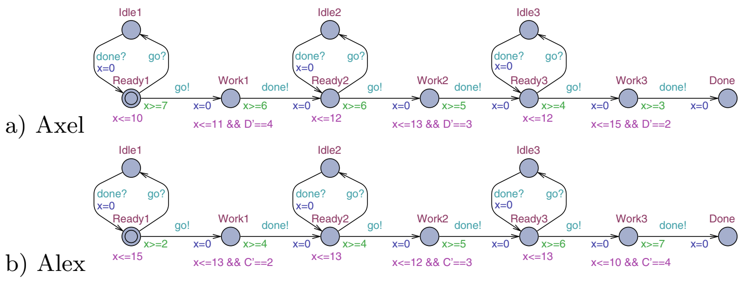


Fig. 17. 3-Nail Hammer Game between Axel and Alex

Though it is clear that Axel has a higher probability of winning than Alex (59% versus 41%) given unbounded time, declaring the competition a draw if it has not finished before 50 seconds actually makes Alex the more likely winner. Similarly, Fig. 18b) illustrates the probability of either of the two players winning given an upper bound on energy. With an unlimited amount of energy, clearly Axel is the most likely winner, whereas limiting the consumption of energy to maximum 52 "energy-units" gives Alex an advantage.

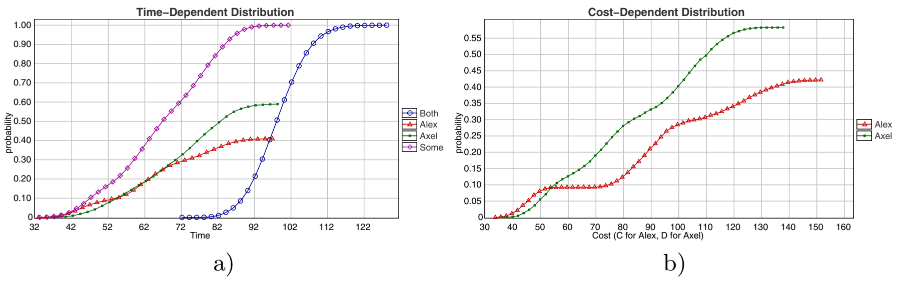


Fig. 18. Time- and Cost-dependent Probability of winning the Hammer Game

5.2 Verifying Queries Using Statistical Model Checking

Following [95], the measure P_{A} may be extended in a standard and unique way to the σ-algebra generated by the sets of runs (so-called cylinders) $\pi(\mathbf{s}, a_1 a_2 \dots a_n)$. As we shall see this will allow us to give proper semantics to a range of probabilistic time- and cost-constrained temporal properties. Let $\mathcal{A}$ be a NPTA. Then we consider the following non-nested PWCTL properties:

$$\psi \ ::= \ \mathsf{P}\big(\Diamond_{C \leq c}\varphi\big) \sim p \ \mid \ \mathsf{P}\big(\Box_{C \leq c}\varphi\big) \sim p$$

where C is an observer clock (of $\mathcal{A}$), φ a state-property (wrt. $\mathcal{A}$) , $\sim \in \{<, \leq, =, \geq, >\}$, and $p \in [0,1]$. This logic is a stochastic extension of the classical WCTL logic for non-stochastic systems, where the existential quantifier is replaced by a probability operator. For the semantics let $\mathcal{A}^*$ be the modification of $\mathcal{A}$, where the guard $C \leq c$ has been conjoined to the invariant of all locations and an edge $(\ell, \varphi, o_\varphi, \emptyset, \ell)$ has been added to all locations ℓ, where o_φ is a new output action. Then:

$$\mathcal{A} \models \mathsf{P}\big(\Diamond_{C \leq c}\varphi\big) \sim p \ \text{ iff } \ \mathsf{P}_{\mathcal{A}^*}\Big(\bigcup_{\sigma \in \Sigma^*} \pi(s_0, \sigma o_\varphi)\Big) \sim p$$

which is well-defined since the σ-algebra on which $\mathsf{P}_{\mathcal{A}^*}$ is defined is closed under countable unions and finite intersections. To complete the semantics, we note that $\mathsf{P}(\Box_{C \leq c}\varphi) \sim p$ is equivalent to $(1-p) \sim \mathsf{P}(\Diamond_{C \leq c}\neg\varphi)$.[4]

[4] We also note that the above (stochastic) interpretation of PWCTL is a conservative extension of the classical (non-stochastic) interpretation of WCTL, in the sense that $\mathcal{A} \models \mathsf{P}\big(\Diamond_{C \leq c}\varphi\big) > 0$ implies $\mathcal{A}_n \models \mathsf{E}\Diamond_{C \leq c}\varphi$, where $\mathcal{A}_n$ refers to the standard non-stochastic semantics of $\mathcal{A}$.

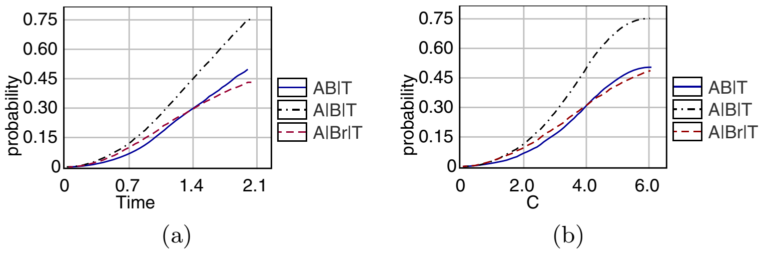


Fig. 19. Cumulative probabilities for time and cost-bounded reachability of T_3

Compared with previous stochastic semantics of timed automata (see e.g., [14,30]), we emphasize the novelty of the semantics of NPTA in terms of RACES between components, truthfully reflecting their independencies. In particular our stochastic semantics of a network $(A_1|..|A_n)$ is significantly different from that obtained by applying the stochastic semantics of [14, 30] to a product construction $A_1A_2 \ldots A_n$, as information about independencies are lost. So though $(A_1|..|A_n)$ and $A_1A_2 \ldots A_n$ are timed bisimilar they are in general not probabistic timed bisimilar, and hence distinguishable by PWCTL. The situation is illustrated with the following example.

Example 9. Reconsider the Example of Fig. 15. Then it can be shown that $(A|B|T) \models \mathsf{P}\big(\Diamond_{t\leq 2}T_3\big) = 0.75$ and $(A|B|T) \models \mathsf{P}\big(\Diamond_{C\leq 6}T_3\big) = 0.75$, whereas $(AB|T) \models \mathsf{P}\big(\Diamond_{t\leq 2}T_3\big) = 0.50$ and $(AB|T) \models \mathsf{P}\big(\Diamond_{C\leq 6}T_3\big) = 0.50$. Fig. 19 gives a time- and cost-bounded reachability probabilities for $(A|B|T)$ and $(AB|T)$ for a range of bounds. Thus, though the two NPTAs satisfy the same WCTL properties, they are obviously quite different with respect to PWCTL. The NPTA B_r of Fig. 15 is a variant of B, with the uniform delay distribution enforced by the invariant $y \leq 2$ being replaced by an exponential distribution with rate $\frac{1}{2}$. Here $(A|B_r|T)$ satisfies $\mathsf{P}\big(\Diamond_{t\leq 2}T_3\big) \approx 0.41$ and $\mathsf{P}\big(\Diamond_{C\leq 6}T_3\big) \approx 0.49$.

The problem of checking $\mathsf{P}_\mathsf{M}(\psi) \geq p$ $(p \in [0,1])$ for a PWCTL property ψ is unfortunately undecidable in general[5]. Our solution is to approximate the answer using simulation-based algorithms known under the name of statistical model checking algorithms. We briefly recap statistical algorithms permitting to answer the following three types of questions:

1. *Hypothesis Testing:* Is the probability $\mathsf{P}_\mathsf{M}(\psi)$ for a given NPTA M greater or equal to a certain threshold $p \in [0,1]$?
2. *Probability evaluation:* What is the probability $\mathsf{P}_M(\psi)$ for a given NPTA M?
3. *Probability comparison:* Is the probability $\mathsf{P}_M(\psi_1)$ greater than the probability $\mathsf{P}_M(\psi_2]$?

[5] Exceptions being PTA with 0 or 1 clocks.

From a conceptual point of view solving the above questions using SMC is simple. First, each run of the system is encoded as a Bernoulli random variable that is true if the run satisfies the property and false otherwise. Then a statistical algorithm groups the observations to answer the three questions. For the qualitative questions (1 and 3), we shall use sequential hypothesis testing, while for the quantitative question (2) we will use an estimation algorithm that resemble the classical Monte Carlo simulation. The two solutions are detailed hereafter.

Hypothesis Testing. This approach reduces the qualitative question to testing the hypothesis $H : p = \mathsf{P}_{\mathsf{M}}(\psi) \geq \theta$ against $K : p < \theta$. To bound the probability of making errors, we use strength parameters α and β and we test the hypothesis $H_0 : p \geq p_0$ and $H_1 : p \leq p_1$ with $p_0 = \theta + \delta_0$ and $p_1 = \theta - \delta_1$. The interval $p_0 - p_1$ defines an indifference region, and p_0 and p_1 are used as thresholds in the algorithm. The parameter α is the probability of accepting H_0 when H_1 holds (false positives) and the parameter β is the probability of accepting H_1 when H_0 holds (false negatives). The above test can be solved by using Wald's *sequential hypothesis testing* [104]. This test computes a proportion r among those runs that satisfy the property. With probability 1, the value of the proportion will eventually cross $\log(\beta/(1-\alpha)$ or $\log((1-\beta)/\alpha)$ and one of the two hypothesis will be selected.

Probability Estimation. This algorithm [70] computes the number of runs needed in order to produce an approximation interval $[p-\varepsilon, p+\varepsilon]$ for $p = Pr(\psi)$ with a confidence $1-\alpha$. The values of ε and α are chosen by the user and the number of runs relies on the Chernoff-Hoeffding bound.

Probability Comparison. This algorithm, which is detailed in [53], exploits an extended Wald testing.

5.3 Uppaal-SMC

We have implemented the above model and algorithms in a statistical extension of Uppaal called Uppaal-SMC. In addition to the features exposed above, the tool also proposes a friendly-user interface to plot results of estimating distributions as well as a distributed engine to exploit computer grids. Details on Uppaal-SMC can be found in [53,54]. As an illustration, here is how we translate the SMC queries from previous section in Uppaal-SMC.

- Hypothesis testing: `Pr[`*bound*`](`φ`)>=`p_0, where *bound* defines how to bound the runs. The three ways to bound them are 1) implicitly by time by specifying `<=`M (where M is a positive integer), 2) explicitly by cost with x`<=`M where x is a specific clock, or 3) by number of discrete steps with `#<=`M. In the case of hypothesis testing p_0 is the probability to test for. The formula φ is either `<>` q or `[]` q where q is a state predicate.
- Estimation: `Pr[`*bound*`](`φ`)`
- Comparison: `Pr[`$bound_1$`](`φ_1`)>= Pr[`$bound_2$`](`φ_2`).`

5.4 Some Illustrations

We briefly survey some recent results obtained via Uppaal-SMC.

Robot Control. In [42] we considered a case – explored in [15] – of a robot moving on a two-dimensional grid. We are interested in the probability that the robot reaches its goal location without staying on consecutive fire fields for more than one time unit and on consecutive ice fields for more than two time units. We applied Uppaal-SMC to compute the probability of the robot reaching this, without staying more than x time units in some fixed position.

Bluetooth [92] is a wireless telecommunication protocol using frequency-hopping to cope with interference between the devices in the wireless network. In paper [54] we adopted the model from [56], annotated the model to record the power utilization and evaluated the probability distributions of likely response times and energy consumption.

Lightweight Medium Access Protocol (LMAC) [103]. LMAC is a communication scheduling protocol based on time slot distribution for nodes sharing the same medium. The protocol is designed having wireless sensor networks in mind: it is simple enough to fit on a modest hardware and at the same time robust against topology reconfiguration, minimizing collisions and power consumption. In [53] we showed how collisions can be analyzed and power consumption estimated using statistical model checking techniques.

Computing Nash Equilibrium in Wireless Ad Hoc Networks. One of the important aspects in designing wireless ad-hoc networks is to make sure that a network is robust to the selfish behavior of its participants, i.e. that its configuration satisfies Nash equilibrium (NE). In [43] we proposed an SMC-based algorithm for computing NE for the case when network nodes are modeled by SPTA and an utility function of a single node is equal to a probability that the node will reach its goal.

Energy Aware Buildings. In [52], we considered energy aware buildings. We refer to a recently developed framework including components for layout of builidngs, availability of heaters, climate and user behavious allowing to evaluated different strategies for distributing heaters among rooms in terms of the resulting comfort and energy consumption. To indicate central parts of this framework and the clear advantages of modeling the evoluation of room temperatures with ODEs, we illustrated in [52] the framework with a small instance comprising two rooms with a single shared heater.

Systems Biology. In [51, 52], we extended our model in order to incorporate ODEs. We then showed how the combination of ODEs and SMC allows us to reason on biological oscillations – a problem that is beyond the scope of most existing formal verification techniques. We model a genetic circadian oscillator, which is used to distill the essence of several real circadian oscillators.

Duration Probabilistic Automata. In [53] we compared UPPAAL-SMC to Prism [77] in the context of Duration Probabilistic Automata (DPA) [90]. A Duration Probabilistic Automaton (DPA) is a composition of Simple Duration Probabilistic Automata (SDPA). An SDPA is a linear sequence of tasks that must be performed in a sequential order. Each task is associated with a duration interval which gives the possible durations of the task. The actual duration of the tasks is given by a uniform choice from this interval. The comparison with Prism was made by randomly generating models with a specific number of SDPAs and a specific number of tasks per SDPA and translate these into Prism and UPPAAL models. The queries to the models were *What is the probability of all SDPAs ending within t time units* (Estimation)and *Is the probability that all SDPAs end within t time units greater than* 40% (Hypothesis testing). The value of t is different for each model as it was computed by simulating the system 369 times and represent the value for which at least 60% of the runs finished all their tasks.

References

1. Abdeddaïm, Y., Kerbaa, A., Maler, O.: Task graph scheduling using timed automata. In: IPDPS, p. 237. IEEE Computer Society (2003)
2. Aceto, L., Ingólfsdóttir, A., Larsen, K.G., Srba, J.: Reactive Systems: Modeling, Specification and Verification. Cambridge University Press (2007)
3. Allamigeon, X., Gaubert, S., Goubault, É.: Inferring min and max invariants using max-plus polyhedra. In: Alpuente, M., Vidal, G. (eds.) SAS 2008. LNCS, vol. 5079, pp. 189–204. Springer, Heidelberg (2008)
4. Alur, R., Courcoubetis, C., Halbwachs, N., Henzinger, T.A., Ho, P.-H., Nicollin, X., Olivero, A., Sifakis, J., Yovine, S.: The algorithmic analysis of hybrid systems. Theoretical Computer Science 138(1), 3–34 (1995)
5. Alur, R.: Timed automata. In: Halbwachs, Peled (eds.) [66], pp. 8–22
6. Alur, R., Bernadsky, M., Madhusudan, P.: Optimal reachability for weighted timed games. In: Díaz, J., Karhumäki, J., Lepistö, A., Sannella, D. (eds.) ICALP 2004. LNCS, vol. 3142, pp. 122–133. Springer, Heidelberg (2004)
7. Alur, R., Courcoubetis, C., Dill, D.L.: Model-checking for real-time systems. In: LICS, pp. 414–425. IEEE Computer Society (1990)
8. Alur, R., Dill, D.L.: Automata for modeling real-time systems. In: Paterson, M. (ed.) ICALP 1990. LNCS, vol. 443, pp. 322–335. Springer, Heidelberg (1990)
9. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. 126(2), 183–235 (1994)
10. Alur, R., La Torre, S., Pappas, G.J.: Optimal paths in weighted timed automata. In: Benedetto, Sangiovanni-Vincentelli (eds.) [26], pp. 49–62
11. Andersson, D.: Improved combinatorial algorithms for discounted payoff games. Master's thesis, Uppsala University, Department of Information Technology (2006)
12. Asarin, E., Bozga, M., Kerbrat, A., Maler, O., Pnueli, A., Rasse, A.: Data-structures for the verification of timed automata. In: Maler, O. (ed.) HART 1997. LNCS, vol. 1201, pp. 346–360. Springer, Heidelberg (1997)
13. Asarin, E., Maler, O., Pnueli, A.: Symbolic controller synthesis for discrete and timed systems. In: Antsaklis, P.J., Kohn, W., Nerode, A., Sastry, S. (eds.) HS 1994. LNCS, vol. 999, pp. 1–20. Springer, Heidelberg (1995)

14. Baier, C., Bertrand, N., Bouyer, P., Brihaye, T., Größer, M.: Probabilistic and topological semantics for timed automata. In: Arvind, V., Prasad, S. (eds.) FSTTCS 2007. LNCS, vol. 4855, pp. 179–191. Springer, Heidelberg (2007)
15. Barbot, B., Chen, T., Han, T., Katoen, J.-P., Mereacre, A.: Efficient CTMC model checking of linear real-time objectives. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 128–142. Springer, Heidelberg (2011)
16. Behrmann, G., Bengtsson, J., David, A., Larsen, K.G., Pettersson, P., Yi, W.: Uppaal implementation secrets. In: Damm, W., Olderog, E.-R. (eds.) FTRTFT 2002. LNCS, vol. 2469, pp. 3–22. Springer, Heidelberg (2002)
17. Behrmann, G., Bouyer, P., Larsen, K.G., Pelánek, R.: Lower and upper bounds in zone based abstractions of timed automata. In: Jensen, Podelski (eds.) [74], pp. 312–326
18. Behrmann, G., Brinksma, E., Hendriks, M., Mader, A.: Production scheduling by reachability analysis - a case study. In: IPDPS. IEEE Computer Society (2005)
19. Behrmann, G., Cougnard, A., David, A., Fleury, E., Larsen, K.G., Lime, D.: Uppaal-Tiga: Time for playing games! In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 121–125. Springer, Heidelberg (2007)
20. Behrmann, G., David, A., Larsen, K.G.: A tutorial on Uppaal. In: Bernardo, M., Corradini, F. (eds.) SFM-RT 2004. LNCS, vol. 3185, pp. 200–236. Springer, Heidelberg (2004)
21. Behrmann, G., Fehnker, A., Hune, T., Larsen, K.G., Pettersson, P., Romijn, J., Vaandrager, F.W.: Minimum-cost reachability for priced timed automata. In: Benedetto, Sangiovanni-Vincentelli (eds.) [26], pp. 147–161
22. Behrmann, G., Hune, T., Vaandrager, F.W.: Distributing timed model checking - how the search order matters. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 216–231. Springer, Heidelberg (2000)
23. Behrmann, G., Larsen, K.G., Pearson, J., Weise, C., Yi, W.: Efficient timed reachability analysis using clock difference diagrams. In: Halbwachs, Peled (eds.) [66], pp. 341–353
24. Behrmann, G., Larsen, K.G., Pelánek, R.: To store or not to store. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 433–445. Springer, Heidelberg (2003)
25. Behrmann, G., Larsen, K.G., Rasmussen, J.I.: Optimal scheduling using priced timed automata. SIGMETRICS Performance Evaluation Review 32(4), 34–40 (2005)
26. Di Benedetto, M.D., Sangiovanni-Vincentelli, A.L. (eds.): HSCC 2001. LNCS, vol. 2034. Springer, Heidelberg (2001)
27. Bengtsson, J.E., Yi, W.: On clock difference constraints and termination in reachability analysis of timed automata. In: Dong, J.S., Woodcock, J. (eds.) ICFEM 2003. LNCS, vol. 2885, pp. 491–503. Springer, Heidelberg (2003)
28. Bengtsson, J.E., Yi, W.: Timed automata: Semantics, algorithms and tools. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) ACPN 2003,. LNCS, vol. 3098, pp. 87–124. Springer, Heidelberg (2004)
29. Bérard, B., Petit, A., Diekert, V., Gastin, P.: Characterization of the expressive power of silent transitions in timed automata. Fundam. Inform. 36(2-3), 145–182 (1998)
30. Bertrand, N., Bouyer, P., Brihaye, T., Markey, N.: Quantitative model-checking of one-clock timed automata under probabilistic semantics. In: QEST, pp. 55–64. IEEE Computer Society (2008)

31. Bohnenkamp, H., D'Argenio, P.R., Hermanns, H., Katoen, J.-P.: Modest: A compositional modeling formalism for real-time and stochastic systems. Technical Report CTIT 04-46, University of Twente (2004)
32. Bouyer, P.: Untameable timed automata! In: Alt, H., Habib, M. (eds.) STACS 2003. LNCS, vol. 2607, pp. 620–631. Springer, Heidelberg (2003)
33. Bouyer, P., Brihaye, T., Markey, N.: Improved undecidability results on weighted timed automata. Inf. Process. Lett. 98(5), 188–194 (2006)
34. Bouyer, P., Brinksma, E., Larsen, K.G.: Staying alive as cheaply as possible. In: Alur, R., Pappas, G.J. (eds.) HSCC 2004. LNCS, vol. 2993, pp. 203–218. Springer, Heidelberg (2004)
35. Bouyer, P., Cassez, F., Fleury, E., Larsen, K.G.: Optimal strategies in priced timed game automata. In: Lodaya, K., Mahajan, M. (eds.) FSTTCS 2004. LNCS, vol. 3328, pp. 148–160. Springer, Heidelberg (2004)
36. Bouyer, P., Fahrenberg, U., Larsen, K.G., Markey, N.: Timed automata with observers under energy constraints. In: HSCC, pp. 61–70. ACM (2010)
37. Bouyer, P., Fahrenberg, U., Larsen, K.G., Markey, N., Srba, J.: Infinite runs in weighted timed automata with energy constraints. In: Cassez, F., Jard, C. (eds.) FORMATS 2008. LNCS, vol. 5215, pp. 33–47. Springer, Heidelberg (2008)
38. Bouyer, P., Larsen, K.G., Markey, N.: Lower-bound constrained runs in weighted timed automata. In: QEST, pp. 128–137. IEEE Computer Society (2012)
39. Bouyer, P., Larsen, K.G., Markey, N., Rasmussen, J.I.: Almost optimal strategies in one clock priced timed games. In: Arun-Kumar, S., Garg, N. (eds.) FSTTCS 2006. LNCS, vol. 4337, pp. 345–356. Springer, Heidelberg (2006)
40. Bozga, M., Daws, C., Maler, O., Olivero, A., Tripakis, S., Yovine, S.: Kronos: A model-checking tool for real-time systems. In: Hu, A.J., Vardi, M.Y. (eds.) CAV 1998. LNCS, vol. 1427, pp. 546–550. Springer, Heidelberg (1998)
41. Brihaye, T., Bruyère, V., Raskin, J.-F.: On optimal timed strategies. In: Pettersson, Yi (eds.) [96], pp. 49–64
42. Bulychev, P., David, A., Larsen, K.G., Legay, A., Li, G., Bøgsted Poulsen, D., Stainer, A.: Monitor-based statistical model checking for weighted metric temporal logic. In: Bjørner, N., Voronkov, A. (eds.) LPAR-18 2012. LNCS, vol. 7180, pp. 168–182. Springer, Heidelberg (2012)
43. Bulychev, P.E., David, A., Larsen, K.G., Legay, A., Mikučionis, M.: Computing Nash equilibrium in wireless ad hoc networks: A simulation-based approach. In: Reich, J., Finkbeiner, B. (eds.) IWIGP. EPTCS, vol. 78, pp. 1–14 (2012)
44. Cassez, F., David, A., Fleury, E., Larsen, K.G., Lime, D.: Efficient on-the-fly algorithms for the analysis of timed games. In: Abadi, M., de Alfaro, L. (eds.) CONCUR 2005. LNCS, vol. 3653, pp. 66–80. Springer, Heidelberg (2005)
45. Cassez, F., David, A., Larsen, K.G., Lime, D., Raskin, J.-F.: Timed control with observation based and stuttering invariant strategies. In: Namjoshi, K.S., Yoneda, T., Higashino, T., Okamura, Y. (eds.) ATVA 2007. LNCS, vol. 4762, pp. 192–206. Springer, Heidelberg (2007)
46. Čerāns, K.: Decidability of bisimulation equivalences for parallel timer processes. In: Probst, D.K., von Bochmann, G. (eds.) CAV 1992. LNCS, vol. 663, pp. 302–315. Springer, Heidelberg (1993)
47. Chatterjee, K., Doyen, L.: Energy parity games. In: Abramsky, S., Gavoille, C., Kirchner, C., Meyer auf der Heide, F., Spirakis, P.G. (eds.) ICALP 2010. LNCS, vol. 6199, pp. 599–610. Springer, Heidelberg (2010)
48. Chatterjee, K., Doyen, L., Henzinger, T.A., Raskin, J.-F.: Generalized mean-payoff and energy games. In: Lodaya, K., Mahajan, M. (eds.) FSTTCS. LIPIcs, vol. 8, pp. 505–516 (2010)

49. Courcoubetis, C., Yannakakis, M.: Minimum and maximum delay problems in real-time systems. In: Larsen, Skou (eds.) [86], pp. 399–409
50. D'Argenio, P.R., Katoen, J.-P., Ruys, T.C., Tretmans, J.: The bounded retransmission protocol must be on time! In: Brinksma, E. (ed.) TACAS 1997. LNCS, vol. 1217, pp. 416–431. Springer, Heidelberg (1997)
51. David, A., Du, D., Larsen, K.G., Legay, A., Mikucionis, M., Poulsen, D.B., Sedwards, S.: Statistical model checking for stochastic hybrid systems. In: HSB. EPTCS, vol. 92, pp. 122–136 (2012)
52. David, A., Larsen, K.G., Legay, A., Mikucionis, M., Poulsen, D.B., Sedwards, S.: Runtime verification of biological systems. In: Margaria, T., Steffen, B. (eds.) ISoLA 2012, Part I. LNCS, vol. 7609, pp. 388–404. Springer, Heidelberg (2012)
53. David, A., Larsen, K.G., Legay, A., Mikučionis, M., Poulsen, D.B., van Vliet, J., Wang, Z.: Statistical model checking for networks of priced timed automata. In: Fahrenberg, U., Tripakis, S. (eds.) FORMATS 2011. LNCS, vol. 6919, pp. 80–96. Springer, Heidelberg (2011)
54. David, A., Larsen, K.G., Legay, A., Mikučionis, M., Wang, Z.: Time for statistical model checking of real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 349–355. Springer, Heidelberg (2011)
55. Dill, D.L.: Timing assumptions and verification of finite-state concurrent systems. In: Sifakis, J. (ed.) CAV 1989. LNCS, vol. 407, pp. 197–212. Springer, Heidelberg (1990)
56. Duflot, M., Kwiatkowska, M., Norman, G., Parker, D.: A formal analysis of bluetooth device discovery. International Journal on Software Tools for Technology Transfer (STTT) 8, 621–632 (2006)
57. Ernits, J.P.: Memory arbiter synthesis and verification for a radar memory interface card. Nord. J. Comput. 12(2), 68–88 (2005)
58. Fahrenberg, U., Juhl, L., Larsen, K.G., Srba, J.: Energy games in multiweighted automata. In: Cerone, A., Pihlajasaari, P. (eds.) ICTAC 2011. LNCS, vol. 6916, pp. 95–115. Springer, Heidelberg (2011)
59. Fahrenberg, U., Larsen, K.G.: Discount-optimal infinite runs in priced timed automata. Electr. Notes Theor. Comput. Sci. 239, 179–191 (2009)
60. Fahrenberg, U., Larsen, K.G.: Discounting in time. Electr. Notes Theor. Comput. Sci. 253(3), 25–31 (2009)
61. Fahrenberg, U., Larsen, K.G., Legay, A., Thrane, C.: Model-based verification, optimization, synthesis and performance evaluation of real-time systems. In: Proceedings of the NATO Advanced Study Institute on Engineering Methods and Tools for Software Safety and Security Marktoberdorf, Germany (August 2012) (to be published 2013)
62. Fahrenberg, U., Larsen, K.G., Thrane, C.: Verification, performance analysis and controller synthesis for real-time systems. In: Broy, M., Sitou, W., Hoare, T. (eds.) ASI 2008. NATO Science for Peace and Security Series - D: Information and Communication Security, vol. 22. IOS Press BV (2008)
63. Fahrenberg, U., Larsen, K.G., Thrane, C.R.: Verification, performance analysis and controller synthesis for real-time systems. In: Arbab, F., Sirjani, M. (eds.) FSEN 2009. LNCS, vol. 5961, pp. 34–61. Springer, Heidelberg (2010)
64. Fahrenberg, U., Larsen, K.G., Thrane, C.: Model-based verification and analysis for real-time systems. In: Broy, M., Leuxner, C., Hoare, T. (eds.) Software and Systems Safety - Specification and Verification. NATO Science for Peace and Security Series - D: Information and Communication Security, vol. 30, pp. 231–259. IOS Press (2011)

65. Fehnker, A.: Scheduling a steel plant with timed automata. In: RTCSA, pp. 280–286. IEEE Computer Society (1999)
66. Halbwachs, N., Peled, D. (eds.): CAV 1999. LNCS, vol. 1633. Springer, Heidelberg (1999)
67. Hansen, M.R., Madsen, J., Brekling, A.W.: Semantics and verification of a language for modelling hardware architectures. In: Jones, C.B., Liu, Z., Woodcock, J. (eds.) Formal Methods and Hybrid Real-Time Systems. LNCS, vol. 4700, pp. 300–319. Springer, Heidelberg (2007)
68. Hendriks, M.: Model checking the time to reach agreement. In: Pettersson, Yi (eds.) [96], pp. 98–111
69. Henzinger, T.A., Nicollin, X., Sifakis, J., Yovine, S.: Symbolic model checking for real-time systems. Inf. Comput. 111(2), 193–244 (1994)
70. Hérault, T., Lassaigne, R., Magniette, F., Peyronnet, S.: Approximate probabilistic model checking. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 73–84. Springer, Heidelberg (2004)
71. Herbreteau, F., Kini, D., Srivathsan, B., Walukiewicz, I.: Using non-convex approximations for efficient analysis of timed automata. In: Chakraborty, S., Kumar, A. (eds.) FSTTCS. LIPIcs, vol. 13, pp. 78–89. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2011)
72. Hune, T., Larsen, K.G., Pettersson, P.: Guided synthesis of control programs using Uppaal. Nord. J. Comput. 8(1), 43–64 (2001)
73. Jensen, H.E., Guldstr, K., Skou, A.: Scaling up Uppaal. In: Joseph, M. (ed.) FTRTFT 2000. LNCS, vol. 1926, pp. 19–30. Springer, Heidelberg (2000)
74. Jensen, K., Podelski, A. (eds.): TACAS 2004. LNCS, vol. 2988. Springer, Heidelberg (2004)
75. Jessen, J.J., Rasmussen, J.I., Larsen, K.G., David, A.: Guided controller synthesis for climate controller using Uppaal Tiga. In: Raskin, J.-F., Thiagarajan, P.S. (eds.) FORMATS 2007. LNCS, vol. 4763, pp. 227–240. Springer, Heidelberg (2007)
76. Karp, R.M.: A characterization of the minimum cycle mean in a digraph. Disc. Math. 23(3), 309–311 (1978)
77. Kwiatkowska, M.Z., Norman, G., Parker, D.: PRISM 2.0: A tool for probabilistic model checking. In: QEST, pp. 322–323. IEEE (2004)
78. Lamport, L.: Real-time model checking is really simple. In: Borrione, D., Paul, W. (eds.) CHARME 2005. LNCS, vol. 3725, pp. 162–175. Springer, Heidelberg (2005)
79. Larsen, K.G., Behrmann, G., Brinksma, E., Fehnker, A., Hune, T., Pettersson, P., Romijn, J.M.T.: As cheap as possible: Efficient cost-optimal reachability for priced timed automata. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 493–505. Springer, Heidelberg (2001)
80. Larsen, K.G., Larsson, F., Pettersson, P., Yi, W.: Efficient verification of real-time systems: compact data structure and state-space reduction. In: IEEE Real-Time Systems Symposium, pp. 14–24. IEEE Computer Society (1997)
81. Larsen, K.G., Mikucionis, M., Nielsen, B., Skou, A.: Testing real-time embedded software using Uppaal-TRON: an industrial case study. In: Wolf, W. (ed.) EMSOFT, pp. 299–306. ACM (2005)
82. Larsen, K.G., Pearson, J., Weise, C., Yi, W.: Clock difference diagrams. Nord. J. Comput. 6(3), 271–298 (1999)
83. Larsen, K.G., Pettersson, P., Yi, W.: Uppaal in a nutshell. STTT 1(1-2), 134–152 (1997)

84. Larsen, K.G., Rasmussen, J.I.: Optimal conditional reachability for multi-priced timed automata. In: Sassone, V. (ed.) FOSSACS 2005. LNCS, vol. 3441, pp. 234–249. Springer, Heidelberg (2005)
85. Larsen, K.G., Rasmussen, J.I.: Optimal reachability for multi-priced timed automata. Theor. Comput. Sci. 390(2-3), 197–213 (2008)
86. Larsen, K.G., Skou, A. (eds.): CAV 1991. LNCS, vol. 575. Springer, Heidelberg (1992)
87. Lindahl, M., Pettersson, P., Yi, W.: Formal design and analysis of a gear controller. In: Steffen, B. (ed.) TACAS 1998. LNCS, vol. 1384, pp. 281–297. Springer, Heidelberg (1998)
88. Lu, Q., Madsen, M., Milata, M., Ravn, S., Fahrenberg, U., Larsen, K.G.: Reachability analysis for timed automata using max-plus algebra. J. Log. Algebr. Program. 81(3), 298–313 (2012)
89. Maler, O.: Timed automata as an underlying model for planning and scheduling. In: Fox, M., Coddington, A.M. (eds.) AIPS Workshop on Planning for Temporal Domains, pp. 67–70 (2002)
90. Maler, O., Larsen, K.G., Krogh, B.H.: On zone-based analysis of duration probabilistic automata. In: INFINITY. EPTCS, vol. 39, pp. 33–46 (2010)
91. Maler, O., Pnueli, A., Sifakis, J.: On the synthesis of discrete controllers for timed systems (an extended abstract). In: Mayr, E.W., Puech, C. (eds.) STACS 1995. LNCS, vol. 900, Springer, Heidelberg (1995)
92. McDermott-Wells, P.: What is bluetooth? IEEE Potentials 23(5), 33–35 (2005)
93. Møller, J.B., Lichtenberg, J., Andersen, H.R., Hulgaard, H.: Difference decision diagrams. In: Flum, J., Rodríguez-Artalejo, M. (eds.) CSL 1999. LNCS, vol. 1683, pp. 111–125. Springer, Heidelberg (1999)
94. Ouaknine, J., Worrell, J.: Universality and language inclusion for open and closed timed automata. In: Maler, O., Pnueli, A. (eds.) HSCC 2003. LNCS, vol. 2623, pp. 375–388. Springer, Heidelberg (2003)
95. Panangaden, P.: Labelled Markov Processes. Imperial College Press (2010)
96. Pettersson, P., Yi, W. (eds.): FORMATS 2005. LNCS, vol. 3829. Springer, Heidelberg (2005)
97. Quaas, K.: On the interval-bound problem for weighted timed automata. In: Dediu, A.-H., Inenaga, S., Martín-Vide, C. (eds.) LATA 2011. LNCS, vol. 6638, pp. 452–464. Springer, Heidelberg (2011)
98. Rasmussen, J.I., Larsen, K.G., Subramani, K.: Resource-optimal scheduling using priced timed automata. In: Jensen, Podelski (eds.) [74], pp. 220–235.
99. Sen, K., Viswanathan, M., Agha, G.: Statistical model checking of black-box probabilistic systems. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 202–215. Springer, Heidelberg (2004)
100. Stirling, C.: Modal and temporal logics for processes. In: Moller, F., Birtwistle, G. (eds.) Logics for Concurrency. LNCS, vol. 1043, pp. 149–237. Springer, Heidelberg (1996)
101. Teige, T., Eggers, A., Fränzle, M.: Constraint-based analysis of concurrent probabilistic hybrid systems: An application to networked automation systems. Nonlinear Analysis: Hybrid Systems (2011)
102. Tripakis, S., Altisen, K.: On-the-fly controller synthesis for discrete and dense-time systems. In: Wing, J.M., Woodcock, J. (eds.) FM 1999. LNCS, vol. 1708, pp. 233–252. Springer, Heidelberg (1999)

103. van Hoesel, L.F.W., Havinga, P.J.M.: A lightweight medium access protocol (LMAC) for wireless sensor networks: Reducing preamble transmissions and transceiver state switches. In: 1st International Workshop on Networked Sensing Systems (INSS 2004), Tokio, Japan, pp. 205–208. Society of Instrument and Control Engineers (SICE) (2004)
104. Wald, A.: Sequential Analysis. Dover Publications (2004)
105. Yi, W., Pettersson, P., Daniels, M.: Automatic verification of real-time communicating systems by constraint-solving. In: Hogrefe, D., Leue, S. (eds.) FORTE. IFIP Conference Proceedings, vol. 6, pp. 243–258. Chapman & Hall (1994)
106. Younes, H.L.S.: Verification and Planning for Stochastic Processes with Asynchronous Events. PhD thesis, Carnegie Mellon (2005)

Unifying Theories of Programming in Isabelle

Simon Foster and Jim Woodcock

Department of Computer Science
University of York
York YO10 5GH
Great Britain
{jim.woodcock,simon.foster}@york.ac.uk
www.cs.york.ac.uk

Abstract. This is a tutorial introduction to the two most basic theories in Hoare & He's Unifying Theories of Programming and their mechanisation in the Isabelle interactive theorem prover. We describe the theories of relations and of designs (pre-postcondition pairs), interspersed with their formalisation in Isabelle and example mechanised proofs.

Keywords: Unifying Theories of Programming (UTP), Denotational Semantics, Laws of Programming, Isabelle, Interactive Theorem Proving.

Dedication: *To Professor He Jifeng on the occasion of his 70th birthday.*

1 Preliminaries

Unifying Theories of Programming, originally the work of Hoare & He [15], is a long-term research agenda that can be summarised as follows. Researchers have proposed many different programming theories and practitioners have proposed many different pragmatic programming paradigms; how do we understand the relationship between them?

UTP can trace its origins back to the work on predicative programming, which was started by Hehner; see [12] for a summary. It gives three principal ways to study such relationships: (i) by computational paradigm; (ii) by level of abstraction; and (iii) by method of presentation.

In Section 2, we introduce the basic concepts of UTP: alphabets, signatures, and healthiness conditions, and in Section 3 we outline the idea of *theory mechanisation* in Isabelle/HOL. In Section 4, we go on to describe the alphabetised relational calculus, the formalism used to describe predicates in UTP theories. In Section 5, we introduce a basic nondeterministic programming language and its laws of programming. In Section 6, we complete the initial presentation of UTP by describing the organisaiton of UTP theories into complete lattices. Sections 7 and 8 show how Hoare logic and the weakest precondition calculus can be defined in UTP. Section 9 introduces the UTP theory of designs that capture the notion of total correctness using assumptions and commitments. The paper ends with a discussion of related work (Section 11) and some conclusions including directions for future work (Section 12).

Computational Paradigms. UTP groups programming languages according to a classification by computational model; for example, structured, object-oriented, functional, or logical. The technique is to identify common concepts and deal separately with additions and variations. It uses two fundamental scientific principles: (i) simplicity of presentation and (ii) separation of concerns.

Abstraction. Orthogonal to this organisation by computational paradigm, languages could be categorised by their level of abstraction within a particular paradigm. For example, the lowest level of abstraction may be the platform-specific technology of an implementation. At the other end of the spectrum, there might be a very high-level description of overall requirements and how they are captured and analysed. In between, there will be descriptions of components and descriptions of how they will be organised into architectures. Each of these levels will have interfaces specified by contracts of some kind. UTP gives ways of mapping between these levels based on a formal notion of refinement that provides guarantees of correctness all the way from requirements to code.

Presentation. The third classification is by the method chosen to present a language definition. There are three widely used scientific methods:

1. *Denotational*, in which each syntactic phrase is given a single mathematical meaning, a specification is just a set of denotations, and refinement is a simple correctness criterion of inclusion: every program behaviour is also a specification behaviour.
2. *Algebraic*, where no direct meaning is given to the language, but instead equalities relate different programs with the same meaning.
3. *Operational* where programs are defined by how they execute on an idealised abstract mathematical machine, giving a useful guide for compilation, debugging, and testing.

As Hoare & He point out [15], a comprehensive account of a programming theory needs all three kinds of presentation, and the UTP technique allows us to study differences and mutual embeddings, and to derive each from the others by mathematical definition, calculation, and proof.

The UTP research agenda has as its ultimate goal to cover all the interesting paradigms of computing, including both declarative and procedural, hardware and software. It presents a theoretical foundation for understanding software and systems engineering, and has been already been exploited in areas such as hardware ([23,39]), hardware/software co-design ([6]) and component-based systems ([38]). But it also presents an opportunity when constructing new languages, especially ones with heterogeneous paradigms and techniques.

Having studied the variety of existing programming languages and identified the major components of programming languages and theories, we can select theories for new, perhaps special-purpose languages. The analogy here is of a theory supermarket, where you shop for exactly those features you need while being confident that the theories plug-and-play together nicely.

A key concept in UTP is the *design*: the familiar precondition-postcondition pair that describes the contract between a programmer and a client. Great use of this construct is made in the semantics of the *Circus* family of languages [35,21], where reactive processes are given a precondition-postcondition semantics that is then useful in assertional reasoning about state-rich reactive behaviour. Parts of this introduction are adapted from [36].

2 Introduction to UTP

The book by Hoare & He [15] sets out a research programme to find a common basis in which to explain a wide variety of programming paradigms: unifying theories of programming (UTP). Their technique is to isolate important language features, and give them a denotational semantics; algebraic and operational semantics can then be proved sound against this model. This allows different languages and paradigms to be compared.

The semantic model is an alphabetised version of Tarski's relational calculus, presented in a predicative style that is reminiscent of the schema calculus in the Z [28,33] notation. Each programming construct is formalised as a relation between an initial and an intermediate or final observation. The collection of these relations forms a *theory* of the paradigm being studied, and it contains three essential parts: an alphabet, a signature, and healthiness conditions.

The alphabet is a set of variable names that gives the vocabulary for the theory being studied. Names are chosen for any relevant external observations of behaviour. For instance, a program with variables x, y, and z would contain these names in its alphabet. Theories for particular programming paradigms require the observation of extra information; some examples are: a flag that says whether the program has started (ok); the current time ($clock$); the number of available resources (res); a trace of the events in the life of the program (tr); a set of refused events (ref); or a flag that says whether the program is waiting for interaction with its environment ($wait$).

The signature gives the rules for the syntax for denoting objects of the theory.

Healthiness conditions identify properties that characterise the predicates of the theory. Each healthiness condition embodies an important fact about the computational model for the programs being studied.

Example 1 (Healthiness conditions (Hoare & He)).

1. The variable *clock* gives us an observation of the current time, which moves ever onwards. The predicate B specifies this.

   $$C \mathrel{\widehat{=}} clock \le clock'$$

   If we add C to the description of some activity, then the variable *clock* describes the time observed immediately before the activity starts, whereas *clock′* describes the time observed immediately after the activity ends. If we suppose that P is a healthy program, then we must have that

$$P \Rightarrow C$$

2. The variable *ok* is used to record whether or not a program has started. A sensible healthiness condition is that we should not observe a program's behaviour until it has started; such programs satisfy the following equation.

$$P \;=\; (ok \Rightarrow P)$$

If the program has not started, its behaviour is not described. □

Healthiness conditions can often be expressed in terms of a function ϕ that makes a program healthy. There is no point in applying ϕ twice, since we cannot make a healthy program even healthier. Therefore, ϕ must be idempotent: $P = \phi(P)$; this equation characterises the healthiness condition.

For example, we can turn the first healthiness condition above into an equivalent equation, $P = P \wedge C$, and then the following function on predicates $and_C \mathrel{\widehat{=}} \lambda X \bullet P \wedge C$ is the required idempotent. □

Example 2 (Boyle's Law).

Alphabet. Consider a simple theory to model the behaviour of a gas with regard to varying temperature and pressure. The physical phenomenon of the behaviour of the gas is subject to Boyle's Law:

> For a fixed amount of an ideal gas kept at a fixed temperature k, p (pressure) and V (volume) are inversely proportional (while one doubles, the other halves).

The alphabet of our theory contains the three mathematical variables described in Boyle's Law: k, p, and V. The model's observations correspond to real-world observations in what we might term *the model-based agenda*: the variables k, p, and V are *shared* with the real world.

Signature. We now need to describe the syntax used to denote objects of the theory. There is a requirement that temperature remains constant, so, to use our model to simulate the effects of Boyle's law, we need two just operations: (i) change the pressure; and (ii) change the volume. This pair of operations form the *signature* of our theory.

Healthiness Conditions. We know the observations we can make of our theory and the two operations we can use to change these observations. We now need to define some *healthiness conditions* as a way of determining membership of the theory. We are interested only in gases that obey Boyle's law, which states that $p * V = k$ must be *invariant.* Healthiness conditions determine the correct states of the system, and here we need both static and dynamic invariants:

- The equation $p * V = k$ is a *static* invariant: it applies to a *state*.
- We also require k to be constant. If we start in the state (k, p, V), where $p * V = k$, then transit to the state (k', p', V'), where $p' * V' = k'$, then we must have that $k' = k$. This is a *dynamic* invariant: it applies to a *relation*.

Some healthiness conditions can be defined using functions. Suppose $\alpha(\phi) = \{p, V, k\}$; then define $\boldsymbol{B}(\phi) = (\exists k \bullet \phi) \wedge (k = p * V)$. Now, regardless of whether ϕ is healthy or not, $\boldsymbol{B}(\phi)$ certainly is. For example:

$$\begin{aligned}
\phi &= (p = 10) \wedge (V = 5) \wedge (k = 100) \\
\boldsymbol{B}(\phi) &= (\exists k \bullet \phi) \wedge (k = p * V) \\
&= (\exists k \bullet (p = 10) \wedge (V = 5) \wedge (k = 100)) \wedge (k = p * V) \\
&= (p = 10) \wedge (V = 5) \wedge (k = p * V) \\
&= (p = 10) \wedge (V = 5) \wedge (k = 50)
\end{aligned}$$

Notice that $\boldsymbol{B}(\boldsymbol{B}(\phi)) = \boldsymbol{B}(\phi)$. This is known as *idempotence*: taking the medicine twice leaves you healthy, no more and no less so than taking the medicine only once. This give us a simple test for healthiness: ϕ is already healthy if applying $\boldsymbol{B}$ leaves it unchanged. That is, if it satisfies the equation $\phi = \boldsymbol{B}(\phi)$. In this sense, ϕ is a fixed point of the idempotent function $\boldsymbol{B}$.

Consider another observation, that the pressure is between 10 and 20Pa:

$$\psi = (p \in 10\,..\,20) \wedge (V = 5)$$

Notice the fact that $\phi \Rightarrow \psi$. Now, if we make both ϕ and ψ healthy, we discover another fact: $\boldsymbol{B}(\phi) \Rightarrow \boldsymbol{B}(\psi)$.

$$\begin{aligned}
&\boldsymbol{B}(\phi) = (p = 10) \wedge (V = 5) \wedge (k = 50) \\
&\boldsymbol{B}(\psi) = (p \in 10\,..\,20) \wedge (V = 5) \wedge (p * V = k) \\
&(p = 10) \wedge (V = 5) \wedge (k = 50) \Rightarrow (p \in 10\,..\,20) \wedge (V = 5) \wedge (p * V = k)
\end{aligned}$$

In fact, $\boldsymbol{B}$ is *monotonic* in the sense that

$$\forall \phi, \psi \bullet (\phi \Rightarrow \psi) \Rightarrow (\boldsymbol{B}(\phi) \Rightarrow \boldsymbol{B}(\psi))$$

The most useful healthiness conditions are *monotonic idempotent functions*, which leads to some very important mathematical properties concerning complete lattices and Galois connections. □

Relations are used as a semantic model for unified languages of specification and programming. Specifications are distinguished from programs only by the fact that the latter use a restricted signature. As a consequence of this restriction, programs satisfy a richer set of healthiness conditions.

Unconstrained relations are too general to handle the issue of program termination; they need to be restricted by healthiness conditions. The result is the theory of designs, which is the basis for the study of the other programming paradigms in [15]. Here, we present the general relational setting, and the transition to the theory of designs.

In the next section, we present the most general theory of UTP: the alphabetised predicates. In the following section, we establish that this theory is a complete lattice. Section 9 restricts the general theory to designs. Next, in Section 10, we present an alternative characterisation of the theory of designs using healthiness conditions. Finally, we conclude with a summary and a brief account of related work.

3 Theory Mechanisation

We have mechanised UTP in the interactive theorem prover *Isabelle/HOL* [19]. This allows the laws of programming to be mechanically verified, and make them available for use in mechnical program derivation, verification and refinement.

Interactive theorem provers (ITPs) have been built as an aid to programmers who wish to prove properties of their programs, such as correctness or refinement. Core to ITPs are proof goals or obligations: ostensible properties which must be discharged by the user under a given set of assumptions. A proof of such a goal consists of a sequence of calculations which transform the assumptions into the goal. The commands used in this transformation are called *proof tactics*, which help the programmer with varying degress of automation.

For instance we may wish to prove the simple property $\exists\, x . x > 6$. In Isabelle we can formalise such a property and proof in the following manner:

theorem *greater-than-six*: $\exists\, x{::}nat.\ x > 6$
  apply (*rule-tac x=7* **in** *exI*)
  apply (*simp*)
done

There are two steps to this simple proof. We first invoke a rule called *exI* which performs *existential introduction*: we explictly supply a value for x for which the property holds, in this case 7. This leaves us with the proof goal $7 > 6$, which can be dispatched by simple arithmetic properties, so we use Isabelle's built in simplifier tactic simp to finish the proof. Isabelle then gives the message *"No subgoals!"*, which means the proof is complete and we can type done. At this point the property *greater-than-six* is entered into the property database for us to use in future proofs.

Mechanised proofs greatly increase the confidence that a given property is true. If we try to prove something which is not correct, Isabelle will not let us. For instance we can try and prove that *all* numbers are greater than six:

theorem *all-greater-than-six*: $\forall\, x{::}nat.\ x > 6$
  apply (*rule-tac allI*)
  — no possible progress

We cannot make much progress with such a proof – there just isn't a tactic to perform this proof as it is incorrect. In fact Isabelle also contains a helpful *counterexample generator* called nitpick [3] which can be used to see if a property can be refuted.

```
theorem all-greater-than-six: ∀ x::nat. x > 6
  nitpick
```

When we run this command Isabelle returns *"Nitpick found a counterexample: $x = 6$"*, which clearly shows why this proof is impossible. We therefore terminate our proof attempt by typing oops. So Isabelle acts as a theoretician's conscience, requiring that properties be comprehensively discharged. Isabelle proofs are also *correct-by-construction.* All proofs in Isabelle are constructed with respect to a small number of axioms in the Isabelle core, even those originating from automated proof tactics. This means that proofs are not predicated on the correctness of the tools and tactics, but only on the correctness of the underlying axioms which makes Isabelle proofs trustworthy.

Such proofs can, however, be tedious for a theortician to construct manually and therefore Isabelle provides a number of *automated proof tactics* to aid in proof. For instance the *greater-than-six* theorem can be proved in one step by application of Isabelle's main automated proof method auto. The auto tactic performs introduction/elimination style classical deduction and simplification in an effort to prove a goal. The user can also extend auto by adding additional rules which it can make use of, increasing the scope of problems which it can deal with.

Additionally, a more recent development is the addition of the sledgehammer [4] tool. Sledgehammer makes use of external first-order *automated theorem provers.* An automated theorem prover (ATP) is a system which can provide solutions to a certain subclass of logical problems. Sledgehammer can make use of a large number of ATPs, such as E [26], Vampire [24], SPASS [32], Waldmeister [13] and Z3 [9]. During a proof the user can invoke sledgehammer which causes the current goal, along with relevant assumptions, to be submitted to the ATPs which attempt a proof. Upon success, a proof command is returned which the user can insert to complete the proof.

For instance, let's say we wish to prove that for any given number there is an even number greater than it. We can prove such a property by calling sledgehammer:

```
theorem greater-than-y-even: ∀ y::nat. ∃ x > y. (x mod 2 = 0)
  sledgehammer
```

In this case, sledgehammer successfully returns with ostensible proofs from four of the ATPs. We can select one of these proofs to see if it works:

```
theorem greater-than-y-even: ∀ y::nat. ∃ x > y. (x mod 2 = 0)
  by (metis Suc-1 even-Suc even-nat-mod-two-eq-zero lessI less-SucI
            numeral-1-eq-Suc-0 numeral-One)
```

The proof command is inserted and successfully discharges the goal, using a total of 7 laws from Isabelle's standard libary. In keeping with with proofs being correct by construction, sledgehammer does not trust the external ATPs to return sound results, but rather uses them as oracles whose proof output must be *reconstructed* with respect to Isabelle's axioms using the internally verified prover metis. So Isabelle is a highly principled theorem prover in which trust can be placed, but also in which a high degree of proof automation can be obtained.

Sledgehammer works particularly well when used in concert with Isabelle's natural language proof script language *Isar*. Isar allows proof to be written in a calculational style familiar to mathematicians. The majority of proofs in tutorial are written Isar, as exemplified in Section 4.

4 The Alphabetised Relational Calculus

The alphabetised relational calculus is similar to Z's schema calculus [28,33], except that it is untyped and somewhat simpler. An *alphabetised predicate* (conventionally written as $P, Q, \ldots, \textbf{\textit{true}}$) is an alphabet-predicate pair, where the predicate's free variables are all members of the alphabet. Relations are predicates in which the alphabet is composed of undecorated variables (x, y, z, ...) and dashed variables (x', a', ...); the former represent initial observations, and the latter, observations made at a later intermediate or final point. The alphabet of an alphabetised predicate P is denoted αP, and may be divided into its before-variables ($in\alpha P$) and its after-variables ($out\alpha P$). A *homogeneous relation* has $out\alpha P = in\alpha P'$, where $in\alpha P'$ is the set of variables obtained by dashing all variables in the alphabet $in\alpha P$. A *condition* ($b, c, d, \ldots, true$) has an empty output alphabet.

Standard predicate calculus operators can be used to combine alphabetised predicates. Their definitions, however, have to specify the alphabet of the combined predicate. For instance, the alphabet of a conjunction is the union of the alphabets of its components: $\alpha(P \wedge Q) = \alpha P \cup \alpha Q$. Of course, if a variable is mentioned in the alphabet of both P and Q, then they are both constraining the same variable.

The alphabetised relational calculus has been mechanised in Isabelle/UTP. An implementation of any calculus in computer science must make decisions about how unspecified details are fleshed out. For Isabelle/UTP this includes concretising the notions of types and values within alphabetised predicates. Isabelle/UTP predicates are parametrically polymorphic in the type of value which variables possess, and the user can supply their own notion of value, with an associated type system and function library. For instance, we are developing a value model for the *VDM* and *CML* specification languages. These will allow users to construct and verify VDM and CML specifications and programs. Indeed it is our hope that any programming language with a well-specified notion of values and types can be reasoned about within the UTP.

An Isabelle/UTP value model consists of four things:

1. A type to represent *values* (α).
2. A type to represent *types* (τ).
3. A *typing relation* ($\_ : \_ :: \alpha \Rightarrow \tau \Rightarrow \mathsf{bool}$), specifies well-typed values.
4. A *definedness predicate* ($\mathcal{D} :: \alpha \Rightarrow \mathsf{bool}$), specifies well-defined values.

Variables within Isabelle/UTP contain a type which specifies the sort of data the variable should point to. The typing relation therefore allows us to realise predicates which are well-typed. The definedness predicate is used to determine when a value has no meaning. For instance it should follow that $\mathcal{D}(x/0) = \mathsf{false}$, whilst $\mathcal{D}1 = \mathsf{true}$. A correct program should never yield undefined values, and this predicate allows us to specify when this is and isn't the case. We omit details of a specific model since this has no effect on the mechanisation of the laws of UTP.

Isabelle/UTP predicates are modelled as sets of *bindings*, where a binding is a mapping from variables to well-typed values. The bindings contained within a predicate are those bindings which make the predicate true. For instance the predicate $x > 5$ is represented by the binding set $\{(x \mapsto 6), (x \mapsto 7), (x \mapsto 8) \cdots\}$. Likewise the predicate true is simply the set of all possible bindings, and false is the empty set $\emptyset$. We then define all the usual operators of predicate calculus which are used in these notes, including $\vee$, $\wedge$, $\neg$, $\Rightarrow$, $\exists$ and $\forall$, most of which map onto binding set operators. An Isabelle/UTP relation is simply a predicate consisting only of dashed and undashed variables.

Isabelle/UTP provides a collection of tactics for aiding in automating proof. The overall aim is to achieve a level of automation such that proof can be at the same level as the standard pen-and-paper proofs contained herein, or even entirely automated. The three main tactics we have developed are as follows:

- utp-pred-tac – predicate calculus reasoning
- utp-rel-tac – relational calculus reasoning
- utp-expr-tac – expression evaluation

These tactics perform *proof by interpretation.* Isabelle/HOL already has a mature library of laws for reasoning about predicates and binary relations. Thus our tactics are designed to make use of these laws by identifying suitable subcalculi within the UTP for which well-known proof procedures exist. The tactics each also have a version in which auto is called after interpretation, for instance utp-pred-auto-tac is simply utp-pred-tac followed by auto. These tactics allow us to easily establish the basic laws of the UTP predicate and relational calculi.

Example 3 (Selection of basic predicate and relational calculus laws).

theorem *AndP-assoc*: '$P \wedge (Q \wedge R)$' = '$(P \wedge Q) \wedge R$'
  by (*utp-pred-tac*)

theorem *AndP-comm*: '$P \wedge Q$' = '$Q \wedge P$'
  by (*utp-pred-auto-tac*)

theorem *AndP-OrP-distr*: '$(P \vee Q) \wedge R$' = '$(P \wedge R) \vee (Q \wedge R)$'
  by (*utp-pred-auto-tac*)

theorem *AndP-contra*: '$P \wedge \neg P$' = *false*
  by (*utp-pred-tac*)

theorem *ImpliesP-export*: '$P \Rightarrow Q$' = '$P \Rightarrow P \wedge Q$'
  by (*utp-pred-tac*)

theorem *SubstP-IffP*: $‘(P \Leftrightarrow Q)[v/x]‘ = ‘P[v/x] \Leftrightarrow Q[v/x]‘$
by (*utp-pred-tac*)

theorem *SemiR-assoc*: $P \;;\; (Q \;;\; R) = (P \;;\; Q) \;;\; R$
by (*utp-rel-auto-tac*)

theorem *SemiR-SkipR-right*: $P \;;\; \mathit{II} = P$
by (*utp-rel-tac*)

Using the tactics we have constructed a large library of algebraic laws for propositional logic and relation algebra. These laws are most easily applied by application of sledgehammer, which will find the most appropriate rules to complete the step of proof. Sledgehammer works particularly well when used in concert with Isabelle's natural language proof script language *Isar.* Isar allows proof to be written in a calculational style familiar to mathematicians. We will cover these in detail in the next section.

5 Laws of Programming

A distinguishing feature of UTP is its concern with program development, and consequently program correctness. A significant achievement is that the notion of program correctness is the same in every paradigm in [15]: in every state, the behaviour of an implementation implies its specification.

If we suppose that $\alpha P = \{a, b, a', b'\}$, then the *universal closure* of P is given simply as $\forall\, a, b, a', b' \bullet P$, which is more concisely denoted as $[\,P\,]$. The correctness of a program P with respect to a specification S is denoted by $S \sqsubseteq P$ (S is refined by P), and is defined as follows.

$$S \sqsubseteq P \quad \textbf{\textit{iff}} \quad [\,P \Rightarrow S\,]$$

Example 4 (Refinement). Suppose we have the specification $x' > x \wedge y' = y$, and the implementation $x' = x + 1 \wedge y' = y$. The implementation's correctness is argued as follows.

$$\begin{aligned} & x' > x \wedge y' = y \;\sqsubseteq\; x' = x + 1 \wedge y' = y && \textit{definition of } \sqsubseteq \\ =\; & [\,x' = x + 1 \wedge y' = y \Rightarrow x' > x \wedge y' = y\,] && \textit{universal one-point rule, twice} \\ =\; & [\,x + 1 > x \wedge y = y\,] && \textit{arithmetic and reflection} \\ =\; & \mathit{true} \end{aligned}$$

And so, the refinement is valid. □

In the following sections, we introduce the definitions of the constructs of a nondeterministic sequential programming language, together with their laws of programming. Each law can be proved correct as a theorem involving the denotational semantics given by its definition. The constructs are: (i) conditional choice; (ii) sequential composition; (iii) assignment; (iv) nondeterminism; and (v) variable blocks.

5.1 Conditional

As a first example of the definition of a programming constructor, we consider conditionals. Hoare & He use an infix syntax for the conditional operator, and define it as follows.

$$
\begin{array}{ll}
P \lhd b \rhd Q \ \widehat{=}\ (b \wedge P) \vee (\neg\, b \wedge Q) & \textbf{\textit{if}}\ \alpha b \subseteq \alpha P = \alpha Q \\
\alpha(P \lhd b \rhd Q) \ \widehat{=}\ \alpha P &
\end{array}
$$

Informally, $P \lhd b \rhd Q$ means P if b else Q.

The presentation of conditional as an infix operator allows the formulation of many laws in a helpful way.

L1	$P \lhd b \rhd P = P$	*idempotence*
L2	$P \lhd b \rhd Q = Q \lhd \neg\, b \rhd P$	*symmetry*
L3	$(P \lhd b \rhd Q) \lhd c \rhd R = P \lhd b \wedge c \rhd (Q \lhd c \rhd R)$	*associativity*
L4	$P \lhd b \rhd (Q \lhd c \rhd R) = (P \lhd b \rhd Q) \lhd c \rhd (P \lhd b \rhd R)$	*distributivity*
L5	$P \lhd \mathit{true} \rhd Q = P = Q \lhd \mathit{false} \rhd P$	*unit*
L6	$P \lhd b \rhd (Q \lhd b \rhd R) = P \lhd b \rhd R$	*unreachable branch*
L7	$P \lhd b \rhd (P \lhd c \rhd Q) = P \lhd b \vee c \rhd Q$	*disjunction*
L8	$(P \odot Q) \lhd b \rhd (R \odot S) = (P \lhd b \rhd R) \odot (Q \lhd b \rhd S)$	*interchange*
L9	$\neg\, (P \lhd b \rhd Q) = (\neg\, P \lhd b \rhd \neg\, Q)$	*cond. neg.*
L10	$(P \lhd b \rhd Q) \wedge \neg\, (R \lhd b \rhd S) = (P \wedge \neg\, R) \lhd b \rhd (Q \wedge \neg\, S)$	*comp.*
L11	$(P \Rightarrow (Q \lhd b \rhd R)) = ((P \Rightarrow Q) \lhd b \rhd (P \Rightarrow R))$	*cond.-⇒-1*
L12	$((P \lhd b \rhd Q) \Rightarrow R) = ((P \Rightarrow R) \lhd b \rhd (Q \Rightarrow R))$	*cond.-⇒-2*
L13	$(P \lhd b \rhd Q) \wedge R = (P \wedge R) \lhd b \rhd (Q \wedge R)$	*cond.-conjunction*
L14	$(P \lhd b \rhd Q) \vee R = (P \vee R) \lhd b \rhd (Q \vee R)$	*cond.-disjunction*
L15	$b \wedge (P \lhd b \rhd Q) = (b \wedge P)$	*cond.-left-simp*
L16	$\neg\, b \wedge (P \lhd b \rhd Q) = (\neg\, b \wedge Q)$	*cond.-right-simp*
L17	$(P \lhd b \rhd Q) = ((b \wedge P) \lhd b \rhd Q)$	*cond.-left*
L18	$(P \lhd b \rhd Q) = (P \lhd b \rhd (\neg\, b \wedge Q))$	*cond.-right*

In the Interchange Law (***L8***), the symbol $\odot$ stands for any truth-functional operator. For each operator, Hoare & He give a definition followed by a number of algebraic laws as those above. These laws can be proved from the definition. As an example, we present the proof of the Unreachable Branch Law (***L6***).

*Example 5 (Proof of Unreachable Branch (**L6**)).*

$$
\begin{array}{ll}
\quad (P \lhd b \rhd (Q \lhd b \rhd R)) & \textbf{\textit{L2}} \\
= ((Q \lhd b \rhd R) \lhd \neg\, b \rhd P) & \textbf{\textit{L3}} \\
= (Q \lhd b \wedge \neg\, b \rhd (R \lhd \neg\, b \rhd P)) & \textit{propositional calculus}
\end{array}
$$

$$= (Q \lhd \mathit{false} \rhd (R \lhd \neg\, b \rhd P)) \qquad \textbf{\textsf{L5}}$$
$$= (R \lhd \neg\, b \rhd P) \qquad \textbf{\textsf{L2}}$$
$$= (P \lhd b \rhd R) \qquad \square$$

This proof can be mechanised in Isar using the same sequence:

*Example 6 (Isar Proof of Unreachable Branch (**L6**)).*

```
theorem CondR-unreachable-branch:
  ‘(P ◁ b ▷ (Q ◁ b ▷ R))‘ = ‘P ◁ b ▷ R‘ (is ?lhs = ?rhs)
proof –
  have ?lhs      = ‘((Q ◁ b ▷ R) ◁ ¬b ▷ P)‘ by (metis CondR-sym)
 also have ... = ‘(Q ◁ b ∧ ¬ b ▷ (R ◁ ¬ b ▷ P))‘ by (metis CondR-assoc)
  also have ... = ‘(Q ◁ false ▷ (R ◁ ¬ b ▷ P))‘ by (utp-pred-tac)
  also have ... = ‘(R ◁ ¬ b ▷ P)‘ by (metis CondR-false)
  also have ... = ?rhs by (metis CondR-sym)
  finally show ?thesis .
qed
```

Isar provides an environment to perform proofs in a natural style, by proving subgoals and the combining these to produce the final goal. The **proof** command opens an Isar proof environment for a goal, and **have** is used to create a subgoal to act as lemma for the overall goal, which must be followed by a proof, usually added using the **by** command. In a calculational proof, we want to transitively compose the previous subgoal with the next, which is done by prefixing the subgoal with **also**. Furthermore Isar provides the ... variable which contains the right-hand side of the previous subgoal. Once all steps of the proof are complete the **finally** command collects all the subgoals together, and **show** is used to prove the overall goal. In the case that no further proof is needed the user can simply type **.** to finish. A completed proof environment can then be terminate with **qed**.

In this case, the proof proceeds by application of sledgehammer for each line where an algebraic law is applied, and by utp-pred-tac when propositional calculus is needed.

Implication is, of course, still the basis for reasoning about the correctness of conditionals. We can, however, prove refinement laws that support a compositional reasoning technique.

Law 51 (Refinement to conditional)

$$P \sqsubseteq (Q \lhd b \rhd R) \;=\; (\, P \sqsubseteq b \wedge Q\,) \wedge (\, P \sqsubseteq \neg\, b \wedge R\,) \qquad \square$$

This result allows us to prove the correctness of a conditional by a case analysis on the correctness of each branch. Its proof is as follows.

Proof of Law 51

$$P \;\sqsubseteq\; (Q \lhd b \rhd R) \qquad \textit{definition of } \sqsubseteq$$
$$= [\,(Q \lhd b \rhd R) \Rightarrow P\,] \qquad \textit{definition of conditional}$$

$$= [\, b \wedge Q \vee \neg\, b \wedge R \Rightarrow P \,] \qquad \textit{propositional calculus}$$
$$= [\, b \wedge Q \Rightarrow P \,] \wedge [\neg\, b \wedge R \Rightarrow P \,] \qquad \textit{definition of } \sqsubseteq\textit{, twice}$$
$$= (\, P \sqsubseteq b \wedge Q \,) \wedge (\, P \sqsubseteq \neg\, b \wedge R \,) \qquad \Box$$

The corresponding proof in Isar can also be given:

Example 7 (Isar Proof of Law 51).

```
theorem RefineP-to-CondR:
  ‘P ⊑ (Q ◁ b ▷ R)‘ = ‘(P ⊑ b ∧ Q) ∧ (P ⊑ ¬ b ∧ R)‘
proof –
  have ‘P ⊑ (Q ◁ b ▷ R)‘ = ‘[(Q ◁ b ▷ R) ⇒ P]‘ by (metis RefP-def)
  also have ... = ‘[(b ∧ Q) ∨ (¬ b ∧ R) ⇒ P]‘ by (metis CondR-def)
  also have ... = ‘[b ∧ Q ⇒ P] ∧ [¬ b ∧ R ⇒ P]‘ by (utp-pred-auto-tac)
  also have ... = ‘(P ⊑ b ∧ Q) ∧ (P ⊑ ¬ b ∧ R)‘ by (metis RefP-def)
  finally show ?thesis .
qed
```

A compositional argument is also available for conjunctions.

Law 52 (Separation of requirements)

$$((P \wedge Q) \sqsubseteq R) = (P \sqsubseteq R) \wedge (Q \sqsubseteq R) \qquad \Box$$

We can prove that an implementation satisfies a conjunction of requirements by considering each conjunct separately. The omitted proof is left as an exception for the interested reader.

5.2 Sequential Composition

Sequence is modelled as relational composition. Two relations may be composed, providing that the output alphabet of the first is the same as the input alphabet of the second, except only for the use of dashes.

$$P(v')\,;\,Q(v) \;\widehat{=}\; \exists\, v_0 \bullet P(v_0) \wedge Q(v_0) \qquad \textbf{\textit{if}}\;\; out\alpha P = in\alpha Q' = \{v'\}$$
$$in\alpha(P(v')\,;\,Q(v)) \;\widehat{=}\; in\alpha P$$
$$out\alpha(P(v')\,;\,Q(v)) \;\widehat{=}\; out\alpha Q$$

Composition is associative and distributes backwards through the conditional.

L1 $P\,;(Q\,;\,R) = (P\,;\,Q)\,;R$ *associativity*

L2 $(P \lhd b \rhd Q)\,;R = ((P\,;\,R) \lhd b \rhd (Q\,;\,R))$ *left distribution*

The simple proofs of these laws, and those of a few others in the sequel, are omitted for the sake of conciseness.

5.3 Assignment

The definition of assignment is basically equality; we need, however, to be careful about the alphabet. If $A = \{x, y, \ldots, z\}$ and $\alpha e \subseteq A$, where αe is the set of free variables of the expression e, the assignment $x :=_A e$ of expression e to variable x changes only x's value.

$$
\begin{aligned}
x :=_A e \ &\widehat{=}\ (x' = e \wedge y' = y \wedge \cdots \wedge z' = z) \\
\alpha(x :=_A e) \ &\widehat{=}\ A \cup A'
\end{aligned}
$$

There is a degenerate form of assignment that changes no variable: it is called "skip", and has the following definition.

$$
\begin{aligned}
\mathit{II}_A \ &\widehat{=}\ (v' = v) && \textbf{\textit{if}}\ A = \{v\} \\
\alpha \mathit{II}_A \ &\widehat{=}\ A \cup A'
\end{aligned}
$$

L1	$(x := e) = (x, y := e, y)$	*framing*
L2	$(x, y, z := e, f, g) = (y, x, z := f, e, g)$	*permutation*
L3	$(x := e \,;\, x := f(x)) = (x := f(e))$	*composition*
L4	$(x := e \,;\, (P \lhd b(x) \rhd Q)) = ((x := e \,;\, P) \lhd b(e) \rhd (x := e \,;\, Q))$	
L5	$P \,;\, \mathit{II}_{\alpha P} = P = \mathit{II}_{\alpha P} \,;\, P$	*unit*
L6	$v' = e \,;\, P \ = \ P[e/v]$ where $\alpha P = \{v, v'\}$	*left-one-point*
L7	$P \,;\, v = e \ = \ P[e/v']$ where $\alpha P = \{v, v'\}$	*right-one-point*

5.4 Nondeterminism

In theories of programming, nondeterminism may arise in one of two ways: either as the result of run-time factors, such as distributed processing; or as the under-specification of implementation choices. Either way, nondeterminism is modelled by choice; the semantics is simply disjunction.

$$
\begin{aligned}
P \sqcap Q \ &\widehat{=}\ P \vee Q && \textbf{\textit{if}}\ \alpha P = \alpha Q \\
\alpha(P \sqcap Q) \ &\widehat{=}\ \alpha P
\end{aligned}
$$

The alphabet must be the same for both arguments.

L1	$P \sqcap Q = Q \sqcap P$	*symmetry*
L2	$P \sqcap (Q \sqcap R) = (P \sqcap Q) \sqcap R$	*associativity*
L3	$P \sqcap P = P$	*idempotence*
L4	$P \sqcap (Q \sqcap R) = (P \sqcap Q) \sqcap (P \sqcap R)$	*distrib.*
L5	$(P \lhd b \rhd (Q \sqcap R)) = (P \lhd b \rhd Q) \sqcap (P \lhd b \rhd R)$	*distrib.*
L6	$(P \sqcap Q) \,;\, R = (P \,;\, R) \sqcap (Q \,;\, R)$	*distrib.*

$$\textbf{\textsf{L7}} \quad P ; (Q \sqcap R) = (P ; Q) \sqcap (P ; R) \qquad \textsf{\textit{distrib.}}$$

$$\textbf{\textsf{L8}} \quad P \sqcap (Q \lhd b \rhd R) = ((P \sqcap Q) \lhd b \rhd (P \sqcap R)) \qquad \textsf{\textit{distrib.}}$$

The following law gives an important property of refinement: if P is refined by Q, then offering the choice between P and Q is immaterial; conversely, if the choice between P and Q behaves exactly like P, so that the extra possibility of choosing Q does not add any extra behaviour, then Q is a refinement of P.

Law 53 (Refinement and nondeterminism)

$$P \sqsubseteq Q \;=\; (P \sqcap Q = P) \qquad \Box$$

Proof

$$\begin{array}{ll}
\quad P \sqcap Q = P & \textsf{\textit{antisymmetry}} \\
= (P \sqcap Q \sqsubseteq P) \wedge (P \sqsubseteq P \sqcap Q) & \textsf{\textit{definition of}} \sqsubseteq\textsf{\textit{, twice}} \\
= [\, P \Rightarrow P \sqcap Q \,] \wedge [\, P \sqcap Q \Rightarrow P \,] & \textsf{\textit{definition of}} \sqcap\textsf{\textit{, twice}} \\
= [\, P \Rightarrow P \vee Q \,] \wedge [\, P \vee Q \Rightarrow P \,] & \textsf{\textit{propositional calculus}} \\
= \mathit{true} \wedge [\, P \vee Q \Rightarrow P \,] & \textsf{\textit{propositional calculus}} \\
= [\, Q \Rightarrow P \,] & \textsf{\textit{definition of}} \sqsubseteq \\
= P \sqsubseteq Q & \Box
\end{array}$$

Another fundamental result is that reducing nondeterminism leads to refinement.

Law 54 (Thin nondeterminism)

$$P \sqcap Q \sqsubseteq P \qquad \Box$$

The proof is immediate from properties of the propositional calculus.

5.5 Alphabet Extension

Alphabet extension is a way adding new variables to the alphabet of a predicate, for example, when new programming variables are declared, as we see in the next section.

$$\begin{array}{lll}
R_{+x} & \mathrel{\widehat=} & R \wedge x' = x \qquad\qquad \text{for} \quad x, x' \notin \alpha R \\
\alpha(R_{+x}) & \mathrel{\widehat=} & \alpha R \cup \{x, x'\}
\end{array}$$

5.6 Variable Blocks

Variable blocks are split into the commands $\textbf{\textit{var}}\, x$, which declares and introduces x in scope, and $\textbf{\textit{end}}\, x$, which removes x from scope. Their definitions are presented below, where A is an alphabet containing x and x'.

$$\begin{array}{llll}
\textbf{\textit{var}}\, x \mathrel{\widehat=} (\exists\, x \bullet I\!I_A) & & \alpha(\textbf{\textit{var}}\, x) \mathrel{\widehat=} A \setminus \{x\} \\
\textbf{\textit{end}}\, x \mathrel{\widehat=} (\exists\, x' \bullet I\!I_A) & & \alpha(\textbf{\textit{end}}\, x) \mathrel{\widehat=} A \setminus \{x'\}
\end{array}$$

The relation $\textbf{\textit{var}}\, x$ is not homogeneous, since it does not include x in its alphabet, but it does include x'; similarly, $\textbf{\textit{end}}\, x$ includes x, but not x'.

The results below state that following a variable declaration by a program Q makes x local in Q; similarly, preceding a variable undeclaration by a program Q makes x' local.

$$(\textbf{\textit{var}}\, x \,;\, Q) = (\exists\, x \bullet Q)$$
$$(Q \,;\, \textbf{\textit{end}}\, x) = (\exists\, x' \bullet Q)$$

More interestingly, we can use ***var*** x and ***end*** x to specify a variable block.

$$(\textbf{\textit{var}}\, x \,;\, Q \,;\, \textbf{\textit{end}}\, x) = (\exists\, x, x' \bullet Q)$$

In programs, we use ***var*** x and ***end*** x paired in this way, but the separation is useful for reasoning.

The following laws are representative.

L6 $(\textbf{\textit{var}}\, x \,;\, \textbf{\textit{end}}\, x) = \mathbb{I}$

L8 $(x := e \,;\, \textbf{\textit{end}}\, x) = (\textbf{\textit{end}}\, x)$

Variable blocks introduce the possibility of writing programs and equations like that below.

$$\begin{aligned} &(\textbf{\textit{var}}\ x \,;\, x := 2 * y \,;\, w := 0 \,;\, \textbf{\textit{end}}\ x) \\ &\qquad = \ (\textbf{\textit{var}}\ x \,;\, x := 2 * y \,;\, \textbf{\textit{end}}\ x) \,;\, w := 0 \end{aligned}$$

Clearly, the assignment to w may be moved out of the scope of the the declaration of x, but what is the alphabet in each of the assignments to w? If the only variables are w, x, and y, and suppose that $A = \{w, y, w', y'\}$, then the assignment on the right has the alphabet A; but the alphabet of the assignment on the left must also contain x and x', since they are in scope. There is an explicit operator for making alphabet modifications such as this: *alphabet extension.* If the right-hand assignment is $P \mathrel{\widehat{=}} w :=_A 0$, then the left-hand assignment is denoted by P_{+x}.

$$\begin{aligned} P_{+x} &\mathrel{\widehat{=}} P \wedge x' = x && \text{for } \ x, x' \notin \alpha P \\ \alpha(P_{+x}) &\mathrel{\widehat{=}} \alpha P \cup \{x, x'\} \end{aligned}$$

If Q does not mention x, then the following laws hold.

L1 $\textbf{\textit{var}}\, x \,;\, Q_{+x} \,;\, P \,;\, \textbf{\textit{end}}\, x \ = \ Q \,;\, \textbf{\textit{var}}\, x \,;\, P \,;\, \textbf{\textit{end}}\, x$

L2 $\textbf{\textit{var}}\, x \,;\, P \,;\, Q_{+x} \,;\, \textbf{\textit{end}}\, x \ = \ \textbf{\textit{var}}\, x \,;\, P \,;\, \textbf{\textit{end}}\, x \,;\, Q$

Together with the laws for variable declaration and undeclaration, the laws of alphabet extension allow for program transformations that introduce new variables and assignments to them.

6 The Complete Lattice

A lattice is a partially ordered set where all non-empty finite subsets have both a least upper-bound (join) and a greatest lower-bound (meet). A complete lattice additionally requires all subsets have both a join and a meet.

Example 8 (Complete lattice: Powerset). The powerset of any set S, ordered by inclusion, forms a complete lattice. The empty set is the least element and S itself is the greatest element. Set union is the join operation and set intersection is the meet. For example, the powerset of $\{0, 1, 2, 3\}$ ordered by subset, is illustrated in Figure 1. □

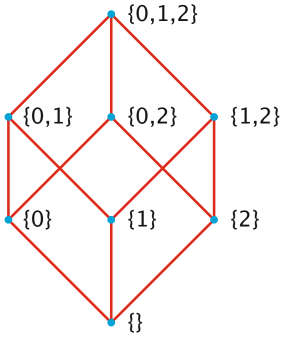


Fig. 1. 0 . . 3 ordered by inclusion

Example 9 (Complete lattice: Divisible natural numbers). Natural numbers ordered by divisibility form a complete lattice. Natural number n is exactly divisible by another natural number m, providing n is an exact multiple of m. This gives us the following partial order:

$$m \sqsubseteq n \Leftrightarrow (\exists k \bullet k \times m = n)$$

In this ordering, 1 is the bottom element (it exactly divides every other number) and 0 is the top element (it can be divided exactly by every other number). For example, if we restrict our attention to the numbers between 0 and 1, we obtain the lattice illustrated in Figure 2. □

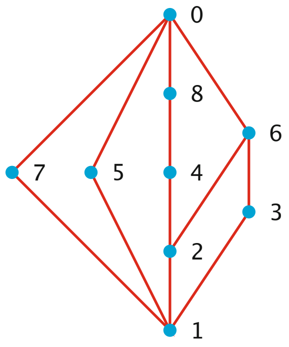


Fig. 2. 0 . . 8 ordered by divisibility

Isabelle/HOL contains a comprehensive mechanised theory of complete lattices and fixed-points, which we directly make use of in Isabelle/UTP. We therefore omit details of these proofs' mechanisation; the reader can refer directly to the HOL library.

6.1 Lattice Operators

The refinement ordering is a partial order: reflexive, anti-symmetric, and transitive. Moreover, the set of alphabetised predicates with a particular alphabet A is a complete lattice under the refinement ordering. Its bottom element is denoted $\bot_A$, and is the weakest predicate ***true***; this is the program that aborts, and behaves quite arbitrarily. The top element is denoted $\top^A$, and is the strongest predicate ***false***; this is the program that performs miracles and implements every specification. These properties of abort and miracle are captured in the following two laws, which hold for all P with alphabet A.

L1	$\bot_A \sqsubseteq P$	*bottom element*
L2	$P \sqsubseteq \top_A$	*top element*

The least upper bound is not defined in terms of the relational model, but by the law ***L1*** below. This law alone is enough to prove laws ***L1A*** and ***L1B***, which are actually more useful in proofs.

L1	$P \sqsubseteq (\bigsqcap S)$ ***iff*** $(P \sqsubseteq X$ ***for all*** X ***in*** $S)$	*unbounded nondeterminism*
L1A	$(\bigsqcap S) \sqsubseteq X$ ***for all*** X ***in*** S	*lower bound*
L1B	***if*** $P \sqsubseteq X$ ***for all*** X ***in*** S, ***then*** $P \sqsubseteq (\bigsqcap S)$	*greatest lower bound*
L2	$(\bigsqcup S) \sqcap Q = \bigsqcup\{ P \sqcap Q \mid P \in S \}$	
L3	$(\bigsqcap S) \sqcup Q = \bigsqcap\{ P \sqcup Q \mid P \in S \}$	
L4	$(\bigsqcap S) \,;\, Q = \bigsqcap\{ P \,;\, Q \mid P \in S \}$	
L5	$R \,;\, (\bigsqcap S) = \bigsqcap\{ R \,;\, P \mid P \in S \}$	

These laws characterise basic properties of least upper bounds.

As we saw above, a function F is *monotonic* if and only if $P \sqsubseteq Q \Rightarrow F(P) \sqsubseteq F(Q)$. Operators like conditional and sequence are monotonic; negation and conjunction are not. There is a class of operators that are all monotonic.

Example 10 (Disjunctivity and monotonicity). Suppose that $P \sqsubseteq Q$ and that $\odot$ is disjunctive, or rather, $R \odot (S \sqcap T) = (R \odot S) \sqcap (R \odot T)$. From this, we can conclude that $P \odot R$ is monotonic in its first argument.

$$
\begin{aligned}
& P \odot R && \textit{assumption } (P \sqsubseteq Q) \textit{ and Law 53} \\
= {} & (P \sqcap Q) \odot R && \textit{assumption } (\odot \textit{ disjunctive}) \\
= {} & (P \odot R) \sqcap (Q \odot R) && \textit{thin nondeterminism} \\
\sqsubseteq {} & Q \odot R
\end{aligned}
$$

A symmetric argument shows that $P \odot Q$ is also monotonic in its other argument. In summary, disjunctive operators are always monotonic. The converse is not true: monotonic operators are not always disjunctive. □

6.2 Recursion

Since alphabetised relations form a complete lattice, every construction defined solely using monotonic operators has a fixed-point. Even more, a result by Knaster and Tarski, described below, says that the set of fixed-points form a complete lattice themselves. The extreme points in this lattice are often of interest; for example, $\top$ is the strongest fixed-point of $X = P \mathbin{;} X$, and $\bot$ is the weakest.

Example 11 (Complete lattice of fixed points). Consider the function $f(s) = s \cup \{0\}$ restricted to the domain comprising the powerset of $\{0, 1, 2\}$. The complete lattice of fixed points for f is illustrated in Fig 3. □

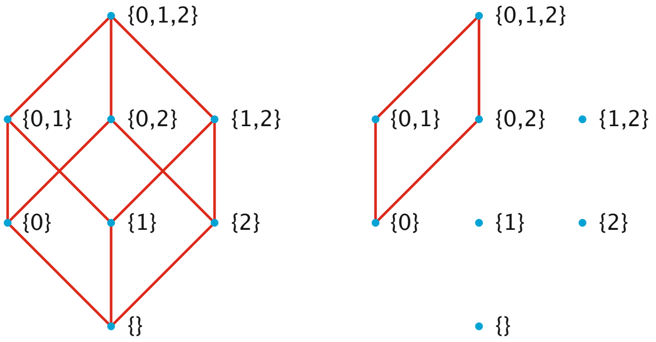


Fig. 3. Complete lattice of fixed points of the function $f(s) = s \cup \{0\}$ (right)

Let $\sqsubseteq$ be a partial order in a lattice X and let $F : X \rightarrow X$ be a function over X. A pre-fixed-point of F is any x such that $F(x) \sqsubseteq x$, and a post-fixed-point of f is any x such that $x \sqsubseteq F(x)$. The Knaster-Tarski theorem states that a monotonic function F on a complete lattice has three properties: (i) The function F has at least one fixed point. (ii) The weakest fixed-point of F coincides with the greatest lower-bound of the set of its post-fixed-points; similarly, the strongest fixed-point coincides with the least upper-bound of the set of its pre-fixed-points. (iii) The set of fixed points of F form a complete lattice.

The weakest fixed-point of the function F is denoted by $\mu\, F$, and is defined simply the greatest lower bound (the *weakest*) of all the pre-fixed-points of F.

$$\mu\, F \;\widehat{=}\; \bigsqcap\{\, X \mid F(X) \sqsubseteq X \,\}$$

The strongest fixed-point νF is the dual of the weakest fixed-point.

Hoare & He use weakest fixed-points to define recursion, where they write a recursive program as $\mu\, X \bullet \mathcal{C}(X)$, where $\mathcal{C}(X)$ is a predicate that is constructed using monotonic operators and the variable X. As opposed to the variables in the alphabet, X stands for a predicate itself, called the recursive variable. Intuitively, occurrences of X in $\mathcal{C}$ stand for recursive calls to $\mathcal{C}$ itself. The definition of recursion is as follows.

$$\mu\, X \bullet \mathcal{C}(X) \;\widehat{=}\; \mu\, F \qquad \textbf{where } F \;\widehat{=}\; \lambda\, X \bullet \mathcal{C}(X)$$

The standard laws that characterise weakest fixed-points are valid.

L1	$\mu\, F \sqsubseteq Y$ if $F(Y) \sqsubseteq Y$	*weakest fixed-point*
L2	$F(\mu\, F) = \mu\, F$	*fixed-point*

L1 establishes that $\mu\, F$ is weaker than any fixed-point; ***L2*** states that $\mu\, F$ is itself a fixed-point. From a programming point of view, ***L2*** is just the copy rule.

Proof of ***L1***

$F(Y) \sqsubseteq Y$	*set comprehension*
$= Y \in \{\, X \mid F(X) \sqsubseteq X \,\}$	*lattice law* ***L1A***
$\Rightarrow \bigsqcap\{\, X \mid F(X) \sqsubseteq X \,\} \sqsubseteq Y$	*definition of* $\mu\, F$
$= \mu\, F \sqsubseteq Y$	□

Proof of ***L2***

$\mu\, F = F(\mu\, F)$	*mutual refinement*
$= \mu\, F \sqsubseteq F(\mu\, F) \wedge F(\mu\, F) \sqsubseteq \mu\, F$	*fixed-point law* ***L1***
$\Leftarrow F(F(\mu\, F)) \sqsubseteq F(\mu\, F) \wedge F(\mu\, F) \sqsubseteq \mu\, F$	F *monotonic*
$\Leftarrow F(\mu\, F) \sqsubseteq \mu\, F$	*definition*
$= F(\mu\, F) \sqsubseteq \bigsqcap\{\, X \mid F(X) \sqsubseteq X \,\}$	*lattice law* ***L1B***
$\Leftarrow \forall\, X \in \{\, X \mid F(X) \sqsubseteq X \,\} \bullet F(\mu f) \sqsubseteq X$	*comprehension*
$= \forall\, X \bullet F(X) \sqsubseteq X \Rightarrow F(\mu\, F) \sqsubseteq X$	*transitivity of* $\sqsubseteq$
$\Leftarrow \forall\, X \bullet F(X) \sqsubseteq X \Rightarrow F(\mu\, F) \sqsubseteq F(X)$	F *monotonic*
$\Leftarrow \forall\, X \bullet F(X) \sqsubseteq X \Rightarrow \mu\, F \sqsubseteq X$	*fixed-point law* ***L1***
$= \mathit{true}$	□

6.3 Iteration

The while loop is written $b * P$: while b is true, execute the program P. This can be defined in terms of the weakest fixed-point of a conditional expression.

$$b * P \;\widehat{=}\; \mu X \bullet ((P \,;\, X) \lhd b \rhd I\!I)$$

Example 12 (Non-termination). If b always remains true, then obviously the loop $b * P$ never terminates, but what is the semantics for this non-termination? The simplest example of such an iteration is $\mathit{true} * I\!I$, which has the semantics $\mu X \bullet X$.

$$\begin{aligned}
&\mu X \bullet X && \textit{definition of least fixed-point}\\
&= \textstyle\bigsqcap\{\, Y \mid (\lambda X \bullet X)(Y) \sqsubseteq Y \,\} && \textit{function application}\\
&= \textstyle\bigsqcap\{\, Y \mid Y \sqsubseteq Y \,\} && \textit{reflexivity of } \sqsubseteq\\
&= \textstyle\bigsqcap\{\, Y \mid \mathit{true} \,\} && \textit{property of } \textstyle\bigsqcap\\
&= \bot && \Box
\end{aligned}$$

$\Box$

A surprising, but simple, consequence of Example 12 is that a program can recover from a non-terminating loop!

Example 13 (Aborting loop). Suppose that the sole state variable is x and that c is a constant.

$$\begin{aligned}
&(b * P);\; x := c && \textit{Example 12}\\
&= \bot;\; x := c && \textit{definition of } \bot\\
&= \textbf{true};\; x := c && \textit{definition of assignment}\\
&= \textbf{true};\; x' = c && \textit{definition of composition}\\
&= \exists\, x_0 \bullet \textbf{true} \land x' = c && \textit{predicate calculus}\\
&= x' = c && \textit{definition of assignment}\\
&= x := c && \Box
\end{aligned}$$

$\Box$

Example 13 is rather disconcerting: in ordinary programming, there is no recovery from a non-terminating loop. It is the purpose of *designs* to overcome this deficiency in the programming model; we return to this in Section 9.

7 Hoare Logic

The Hoare triple $p \{Q\} r$ is a specification of the correctness of a program Q. Here, p and r are assertions and Q is a command. This is partial correctness in the sense that the assertions do not require Q to terminate. Instead, the correctness statement is that, if Q is started in a state satisfying p, then, if it does terminate, it will finish in a state satisfying r. We define the meaning of the Hoare triple as a universal implication:

$$p\,\{Q\}\,r \;\widehat{=}\; [\,p \wedge Q \Rightarrow r'\,]$$

This is a correctness assertion that can be expressed as the refinement assertion $(p \Rightarrow r') \sqsubseteq Q$.

The laws that can be proved from this definition form the Axioms of Hoare Logic:

L1 **if** $p\,\{Q\}\,r$ **and** $p\,\{Q\}\,s$ **then** $p\,\{Q\}\,(r \wedge s)$

L2 **if** $p\,\{Q\}\,r$ **and** $q\,\{Q\}\,r$ **then** $(p \vee q)\,\{Q\}\,r$

L3 **if** $p\,\{Q\}\,r$ **then** $(p \wedge q)\,\{Q\}\,(r \vee s)$

L4 $r(e)\,\{x := e\}\,r(x)$

L5 **if** $(p \wedge b)\,\{Q_1\}\,r$ **and** $(p \wedge \neg\, b)\,\{Q_2\}\,r$ **then**
$p\,\{\,Q_1 \lhd b \rhd Q_2\,\}\,r$

L6 **if** $p\,\{Q_1\}\,s$ **and** $s\,\{Q_2\}\,r$ **then** $p\,\{\,Q_1\,;\,Q_2\,\}\,r$

L7 **if** $p\,\{Q_1\}\,r$ **and** $p\,\{Q_2\}\,r$ **then** $p\,\{\,Q_1 \sqcap Q_2\,\}\,r$

L8 **if** $(b \wedge c)\,\{Q\}\,c$ **then** $c\,\{\,\nu X \bullet (Q\,;\,X) \lhd b \rhd I\!I\,\}\,(\neg\, b \wedge c)$

L9 $\mathit{false}\,\{Q\}\,r$ **and** $p\,\{Q\}\,\mathit{true}$ **and** $p\,\{\textbf{false}\}\,\mathit{false}$ **and** $p\,\{I\!I\}\,p$

Proof of ***L1***.

$$\begin{aligned}
&(p\,\{Q\}\,r) \wedge (p\,\{Q\}\,s)\\
&= (Q \Rightarrow (p \Rightarrow r')) \wedge (Q \Rightarrow (p \Rightarrow s'))\\
&= (Q \Rightarrow (p \Rightarrow r') \wedge (p \Rightarrow s'))\\
&= (Q \Rightarrow (p \Rightarrow r' \wedge s'))\\
&= p\,\{Q\}\,(r \wedge s) && [\Box]
\end{aligned}$$

Proof of ***L8***: Suppose that $(b \wedge c)\,\{Q\}\,c$. Define $Y \;\widehat{=}\; c \Rightarrow \neg\, b' \wedge c'$

$$\begin{aligned}
&c\,\{\,\nu X \bullet (Q\,;\,X) \lhd b \rhd I\!I\,\}\,(\neg\, b \wedge c)\\
&= Y \sqsubseteq \nu X \bullet (Q\,;\,X) \lhd b \rhd I\!I && \textit{by definition}\\
&\Leftarrow Y \sqsubseteq (Q\,;\,Y) \lhd b \rhd I\!I && \textit{by sfp } \textbf{\textit{L1}}\\
&= (\,Y \sqsubseteq (b \wedge Q)\,;\,Y\,) \wedge (\,Y \sqsubseteq \neg\, b \wedge I\!I\,) && \textit{refinement to cond}\\
&= (\,Y \sqsubseteq (b \wedge Q)\,;\,Y\,) \wedge [\neg\, b \wedge I\!I \Rightarrow (c \Rightarrow \neg\, b' \wedge c')\,] && \textit{by def}\\
&= (\,Y \sqsubseteq (b \wedge Q)\,;\,Y\,) \wedge \mathit{true} && \text{predicate calculus}\\
&= c\,\{\,b \wedge Q\,;\,Y\,\}\,(\neg\, b \wedge c) && \textit{by definition}\\
&\Leftarrow (\,c\,\{\,b \wedge Q\,\}\,c\,) \wedge (\,c\,\{\,c \Rightarrow \neg\, b' \wedge c'\,\}\,\neg\, b \wedge c\,) && \textit{by Hoare } \textbf{\textit{L6}}\\
&= \mathit{true} && \textit{by assumption and predicate calculus}
\end{aligned}$$

$\Box$

8 Weakest Preconditions

A Hoare triple involves three variables: a precondition, a postcondition, and a command. If we fix two of these variables, then we can calculate an extreme solution for the third. For example, if we fix the command and the precondition, then we calculate the strongest postcondition. Alternatively, we could fix the command and the postcondition and calculate the weakest precondition, and that is what we do here. We start with some relational calculus to obtain an implication with the precondition assertion as the antecedent of an implication of the form: $[\, p \Rightarrow R \,]$. If we fix R, then there are perhaps many solutions for p that satisfy this inequality. Of all the possibilities, the weakest must actually be equal to R.

$$\begin{aligned}
& p(v) \, \{ \, Q(v, v') \, \} \, r(v) \\
& = [\, Q(v, v') \Rightarrow (\, p(v) \Rightarrow r(v') \,) \,] \\
& = [\, p(v) \Rightarrow (\, Q(v, v') \Rightarrow r(v') \,) \,] \\
& = [\, p(v) \Rightarrow (\forall\, v' \bullet Q(v, v') \Rightarrow r(v') \,) \,] \\
& = [\, p(v) \Rightarrow \neg\, (\, \exists\, v' \bullet Q(v, v') \land \neg\, r(v') \,) \,] \\
& = [\, p(v) \Rightarrow \neg\, (\, \exists\, v_0 \bullet Q(v, v_0) \land \neg\, r(v_0) \,) \,] \\
& = [\, p(v) \Rightarrow \neg\, (\, Q(v, v') \,;\, \neg\, r(v) \,) \,]
\end{aligned}$$

So now, if we take $\mathcal{W}(v) = \neg\, (\, Q(v, v') \,;\, \neg\, r(v) \,)$, then the following Hoare triple must be valid:

$$\mathcal{W}(v) \, \{ \, Q(v, v') \, \} \, r(v)$$

Here, $\mathcal{W}$ is the weakest solution for the precondition for Q to be guaranteed to achieve r.

We define the predicate transformer ***wp*** as a relation between Q and r as follows:

$$Q \, \textbf{\textit{wp}} \, r \;\widehat{=}\; \neg\, (Q \,;\, \neg\, r)$$

The laws for the weakest precondition operator are as follows:

L1	$((x := e) \, \textbf{\textit{wp}} \, r(x)) = r(e)$
L2	$((P \,;\, Q) \, \textbf{\textit{wp}} \, r) = (P \, \textbf{\textit{wp}}(Q \, \textbf{\textit{wp}} \, r))$
L3	$((P \lhd b \rhd Q) \, \textbf{\textit{wp}} \, r) = ((P \, \textbf{\textit{wp}} \, r) \lhd b \rhd (Q \, \textbf{\textit{wp}} \, r))$
L4	$((P \sqcap Q) \, \textbf{\textit{wp}} \, r) = (P \, \textbf{\textit{wp}} \, r) \land (Q \, \textbf{\textit{wp}} \, r)$
L5	if $[\, r \Rightarrow s \,]$ then $[\, (Q \, \textbf{\textit{wp}} \, r) \Rightarrow (Q \, \textbf{\textit{wp}} \, s) \,]$
L6	if $[\, Q \Rightarrow S \,]$ then $[\, (S \, \textbf{\textit{wp}} \, r) \Rightarrow (Q \, \textbf{\textit{wp}} \, r) \,]$
L7	$(Q \, \textbf{\textit{wp}}(\bigwedge R)) = \bigwedge \{ \, (Q \, \textbf{\textit{wp}} \, r) \mid r \in R \, \}$
L8	$(Q \, \textbf{\textit{wp}} \, \mathit{false}) = \mathit{false}$ ***if*** $Q \,;\, \textbf{\textit{true}} = \textbf{\textit{true}}$

A representative selection of Isabelle proofs for these rules is shown below. Most of them can be proved automatically.

```
theorem $SemiR\text{-}wp$: $(P\ ;\ Q)\ wp\ R = P\ wp\ (Q\ wp\ R)$
  by ($utp\text{-}rel\text{-}auto\text{-}tac$)

theorem $CondR\text{-}wp$:
  assumes
    $(P \in WF\text{-}RELATION)\ (Q \in WF\text{-}RELATION)$
    $(b \in WF\text{-}CONDITION)\ (R \in WF\text{-}RELATION)$
  shows ‘$(P \lhd b \rhd Q)\ wp\ R$‘ = ‘$(P\ wp\ R) \lhd b \rhd (Q\ wp\ R)$‘
proof −
  have ‘$(P \lhd b \rhd Q)\ wp\ R$‘ = ‘$\neg ((P \lhd b \rhd Q)\ ;\ (\neg R))$‘
    by ($simp\ add$: $WeakPrecondP\text{-}def$)
  also from $assms$ have ... = ‘$\neg ((P\ ;\ (\neg R)) \lhd b \rhd (Q\ ;\ (\neg R)))$‘
    by ($simp\ add$:$CondR\text{-}SemiR\text{-}distr\ closure$)
  also have ... = ‘$(P\ wp\ R) \lhd b \rhd (Q\ wp\ R)$‘
    by ($utp\text{-}pred\text{-}auto\text{-}tac$)
  finally show $?thesis$ .
qed

theorem $ChoiceP\text{-}wp$:
  $(P \sqcap Q)\ wp\ R$ = ‘$(P\ wp\ R) \wedge (Q\ wp\ R)$‘
  by ($utp\text{-}rel\text{-}auto\text{-}tac$)

theorem $ImpliesP\text{-}precond\text{-}wp$:
  ‘$[R \Rightarrow S]$‘ $\Longrightarrow$ ‘$[(Q\ wp\ R) \Rightarrow (Q\ wp\ S)]$‘
  by ($metis\ ConjP\text{-}wp\ RefP\text{-}AndP\ RefP\text{-}def\ less\text{-}eq\text{-}WF\text{-}PREDICATE\text{-}def$)

theorem $FalseP\text{-}wp$:
  $Q\ ;\ true = true \Longrightarrow Q\ wp\ false = false$
  by ($simp\ add$: $WeakPrecondP\text{-}def$)
```

9 Designs

The problem pointed out in Section 6—that the relational model does not capture the semantics of nonterminating programs—can be explained as the failure of general alphabetised predicates P to satisfy the equation below.

$$\textbf{\textit{true}}\ ; P \ = \ \textbf{\textit{true}}$$

In particular, in Example 13 we presented a non-terminating loop which, when followed by an assignment, behaves like the assignment. Operationally, it is as though the non-terminating loop could be ignored.

The solution is to consider a subset of the alphabetised predicates in which a particular observational variable, called *ok*, is used to record information about the start and termination of programs. The above equation holds for predicates P in this set. As an aside, we observe that *false* cannot possibly belong to this set, since ***true*** ; ***false*** = ***false***.

The predicates in this set are called designs. They can be split into precondition-postcondition pairs, and are in the same spirit as specification statements used in refinement calculi. As such, they are a basis for unifying languages and methods like B [1], VDM [16], Z [33], and refinement calculi [17,2,18].

In designs, *ok* records that the program has started, and *ok*′ records that it has terminated. These are auxiliary variables, in the sense that they appear in a design's alphabet, but they never appear in code or in preconditions and postconditions.

In implementing a design, we are allowed to assume that the precondition holds, but we have to fulfill the postcondition. In addition, we can rely on the program being started, but we must ensure that the program terminates. If the precondition does not hold, or the program does not start, we are not committed to establish the postcondition nor even to make the program terminate.

A design with precondition P and postcondition Q, for predicates P and Q not containing *ok* or *ok*′, is written $(\, P \vdash Q \,)$. It is defined as follows.

$$(\, P \vdash Q \,) \;\widehat{=}\; (\, ok \land P \Rightarrow ok' \land Q \,)$$

If the program starts in a state satisfying P, then it will terminate, and on termination Q will be true.

Example 14 (Specifications). Suppose that we have two program variables, x and y, and that we want to specify an operation on the state that reduces the value of x but keeps y constant. Furthermore, suppose that x and y take on values that are natural numbers. We could specify this operation in Z using an operation schema[28,33]:

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Dec}} \\ \hline
x, y, x', y' : \mathbb{N} \\ \hline
x > 0 \\
x' < x \\
y' = y \\ \hline
\end{array}$$

This specifies the decrement operation *Dec* involving the state before the operation (x and y) and the state after the operation x' and y'). The value of x must be strictly positive, or else the invariant that x is always a natural number cannot be satisfied. The after-value x' must be strictly less than the before-value x and the value of y is left unchanged. This Z schema defines its operation as a single relation, just like the alphabetised relations already introduced.

The refinement calculus [17] is similar to Z, except that the relation specifying a program is slit into a precondition and a postcondition, with the meaning described above: if the program is activated in a state satisfying the precondition, then the program must terminate and when it does, the postcondition will be true. Our operation is specified like this:

$$Dec \;\widehat{=}\; x : [\, x > 0, x < x_0 \,]$$

Here, the before-value of x in the postcondition is specified as x_0 and the after-value as simply x. The frame, written before the precondition-postcondition pair, specifies that only the variable x may be changed; y must remain constant. Something similar happens in VDM [16]:

```
operation Dec()
ext wr x: Nat
pre    x > 0
post   x < x~
```

In UTP, the same operation is specified using a design; the frame is specified by saying what must remain constant:

$$Dec \mathrel{\widehat{=}} (\, x > 0 \vdash x' < x \,)_{+y}$$

□

9.1 Lattice Operators

Abort and miracle are defined as designs in the following examples. Abort has precondition ***false*** and is never guaranteed to terminate. It is denoted by $\bot_{\mathbf{D}}$.

Example 15 (Abort).

$$\begin{array}{ll}
\quad \textbf{\textit{false}} \vdash \textbf{\textit{false}} & \textit{definition of design} \\
= ok \wedge \textbf{\textit{false}} \Rightarrow ok' \wedge \textbf{\textit{false}} & \textbf{\textit{false}} \textit{ zero for conjunction} \\
= \textbf{\textit{false}} \Rightarrow ok' \wedge \textbf{\textit{false}} & \textit{vacuous implication} \\
= \textbf{\textit{true}} & \textit{vacuous implication} \\
= \textbf{\textit{false}} \Rightarrow ok' \wedge \textbf{\textit{true}} & \textbf{\textit{false}} \textit{ zero for conjunction} \\
= ok \wedge \textbf{\textit{false}} \Rightarrow ok' \wedge \textbf{\textit{true}} & \textit{definition of design} \\
= \textbf{\textit{false}} \vdash \textbf{\textit{true}} & \Box
\end{array}$$

□

Miracle has precondition ***true***, and establishes the impossible: ***false***. It is denoted by $\top_{\mathbf{D}}$.

Example 16 (Miracle).

$$\begin{array}{ll}
\quad \textbf{\textit{true}} \vdash \textbf{\textit{false}} & \textit{definition of design} \\
= ok \wedge \textbf{\textit{true}} \Rightarrow ok' \wedge \textbf{\textit{false}} & \textbf{\textit{true}} \textit{ unit for conjunction} \\
= ok \Rightarrow \textbf{\textit{false}} & \textit{contradiction} \\
= \neg\, ok & \Box
\end{array}$$

□

Like the set of general alphabetised predicates, designs form a complete lattice. We have already presented the top and the bottom (miracle and abort).

$$\begin{array}{lcl}
\top_{\mathbf{D}} & \mathrel{\widehat{=}} & (\, \textbf{\textit{true}} \vdash \textbf{\textit{false}} \,) = \neg\, ok \\
\bot_{\mathbf{D}} & \mathrel{\widehat{=}} & (\, \textbf{\textit{false}} \vdash \textbf{\textit{true}} \,) = \textbf{\textit{true}}
\end{array}$$

The least upper bound and the greatest lower bound are established in the following theorem.

Theorem 1. *Meets and joins*

$$\bigsqcap_i (\, P_i \vdash Q_i \,) = (\, \bigwedge_i P_i \,) \vdash (\, \bigvee_i Q_i \,)$$
$$\bigsqcup_i (\, P_i \vdash Q_i \,) = (\, \bigvee_i P_i \,) \vdash (\, \bigwedge_i P_i \Rightarrow Q_i \,)$$

As with the binary choice, the choice $\bigsqcap_i (\, P_i \vdash Q_i \,)$ terminates when all the designs do, and it establishes one of the possible postconditions. The least upper bound models a form of choice that is conditioned by termination: only the terminating designs can be chosen. The choice terminates if any of the designs does, and the postcondition established is that of any of the terminating designs.

Example 17 (Not a design). Notice that designs are not closed under negation.

$$\begin{aligned}
& \neg\, (P \vdash Q) && \textsf{\textit{design}} \\
= {} & \neg\, (ok \wedge p \Rightarrow ok' \wedge Q) && \textsf{\textit{propositional calculus}} \\
= {} & ok \wedge p \wedge (ok' \Rightarrow \neg\, Q)
\end{aligned}$$

Although the negation of a design is not itself a design, this derivation does give a useful identity. □

9.2 Refinement of Designs

A reassuring result about a design is the fact that refinement amounts to either weakening the precondition, or strengthening the postcondition in the presence of the precondition. This is established by the result below.

Law 91 (Refinement of designs)

$$P_1 \vdash Q_1 \sqsubseteq P_2 \vdash Q_2 \;=\; [\, P_1 \wedge Q_2 \Rightarrow Q_1 \,] \wedge [\, P_1 \Rightarrow P_2 \,] \qquad \Box$$

Proof

$$\begin{aligned}
& P_1 \vdash Q_1 \;\sqsubseteq\; P_2 \vdash Q_2 && \textsf{\textit{definition of}} \sqsubseteq \\
= {} & [\, (\, P_2 \vdash Q_2 \,) \Rightarrow (\, P_1 \vdash Q_1 \,) \,] && \textsf{\textit{definition of design, twice}} \\
= {} & [\, (\, ok \wedge P_2 \Rightarrow ok' \wedge Q_2 \,) \Rightarrow (\, ok \wedge P_1 \Rightarrow ok' \wedge Q_1 \,) \,] \\
& && \textsf{\textit{case analysis on }} ok \\
= {} & [\, (\, P_2 \Rightarrow ok' \wedge Q_2 \,) \Rightarrow (\, P_1 \Rightarrow ok' \wedge Q_1 \,) \,] && \textsf{\textit{case analysis on }} ok' \\
= {} & [\, (\, (\, P_2 \Rightarrow Q_2 \,) \Rightarrow (\, P_1 \Rightarrow Q_1 \,) \,) \wedge (\, \neg\, P_2 \Rightarrow \neg\, P_1 \,) \,] && \textsf{\textit{propositional calculus}} \\
= {} & [\, (\, (\, P_2 \Rightarrow Q_2 \,) \Rightarrow (\, P_1 \Rightarrow Q_1 \,) \,) \wedge (\, P_1 \Rightarrow P_2 \,) \,] && \textsf{\textit{predicate calculus}} \\
= {} & [\, P_1 \wedge Q_2 \Rightarrow Q_1 \,] \wedge [\, P_1 \Rightarrow P_2 \,] && \Box
\end{aligned}$$

9.3 Nontermination

The most important result, however, is that abort is a zero for sequence. This was, after all, the whole point for the introduction of designs.

L1 $\textbf{\textit{true}} \; ; (P \vdash Q) = \textbf{\textit{true}}$ *left-zero*

Proof

$$
\begin{array}{ll}
\quad \textbf{\textit{true}} \; ; (P \vdash Q) & \textit{property of sequential composition} \\
= \exists\, ok_0 \bullet \textbf{\textit{true}} \; ; (P \vdash Q)[ok_0/ok] & \textit{case analysis} \\
= (\, \textbf{\textit{true}} \; ; (P \vdash Q)[true/ok] \,) \lor (\, \textbf{\textit{true}} \; ; (P \vdash Q)[false/ok] \,) & \textit{property of design} \\
= (\, \textbf{\textit{true}} \; ; (P \vdash Q)[true/ok] \,) \lor (\, \textbf{\textit{true}} \; ; \textbf{\textit{true}} \,) & \textit{relational calculus} \\
= (\, \textbf{\textit{true}} \; ; (P \vdash Q)[true/ok] \,) \lor \textbf{\textit{true}} & \textit{propositional calculus} \\
= \textbf{\textit{true}} & \Box
\end{array}
$$

9.4 Assignment

In this new setting, it is necessary to redefine assignment and skip, as those introduced previously are not designs.

$$(x := e) \;\widehat{=}\; (\, \textbf{\textit{true}} \vdash x' = e \land y' = y \land \cdots \land z' = z \,)$$

$$I\!I_{D} \;\widehat{=}\; (\, \textbf{\textit{true}} \vdash I\!I \,)$$

Their existing laws hold, but it is necessary to prove them again, as their definitions changed.

L2 $(v := e \; ; v := f(v)) = (v := f(e))$

L3 $(v := e \; ; (P \lhd b(v) \rhd Q)) = ((v := e \; ; P) \lhd b(e) \rhd (v := e \; ; Q))$

L4 $(I\!I_{D} \; ; (P \vdash Q)) = (P \vdash Q)$

As as an example, we present the proof of ***L2***.

Proof of ***L2***

$$
\begin{array}{ll}
\quad v := e \; ; v := f(v) & \textit{definition of assignment, twice} \\
= (\, \textbf{\textit{true}} \vdash v' = e \,) \; ; (\, \textbf{\textit{true}} \vdash v' = f(v) \,) & \textit{case analysis on } ok_0 \\
= (\, (\, \textbf{\textit{true}} \vdash v' = e \,)[true/ok'] \; ; (\, \textbf{\textit{true}} \vdash v' = f(v) \,)[true/ok] \,) \lor & \\
\quad \neg\, ok \; ; \textbf{\textit{true}} & \textit{definition of design} \\
= (\, (\, ok \Rightarrow v' = e \,) \; ; (\, ok' \land v' = f(v) \,) \,) \lor \neg\, ok & \textit{relational calculus} \\
= ok \Rightarrow (\, v' = e \; ; (\, ok' \land v' = f(v) \,) \,) & \textit{assignment composition} \\
= ok \Rightarrow ok' \land v' = f(e) & \textit{definition of design} \\
= (\, \textbf{\textit{true}} \vdash v' = f(e) \,) & \textit{definition of assignment} \\
= v := f(e) & \Box
\end{array}
$$

9.5 Closure under the Program Combinators

If any of the program operators are applied to designs, then the result is also a design. This follows from the laws below, for choice, conditional, sequence, and recursion. The choice between two designs is guaranteed to terminate when they both terminate; since either of them may be chosen, then either postcondition may be established.

$$\textbf{\textit{T1}}\quad (\,(P_1 \vdash Q_1) \sqcap (P_2 \vdash Q_2)\,) = (\,P_1 \wedge P_2 \vdash Q_1 \vee Q_2\,)$$

If the choice between two designs depends on a condition b, then so do the precondition and the postcondition of the resulting design.

$$\begin{aligned}\textbf{\textit{T2}}\quad & (\,(P_1 \vdash Q_1) \lhd b \rhd (P_2 \vdash Q_2)\,)\\ & = ((P_1 \lhd b \rhd P_2) \vdash (\,Q_1 \lhd b \rhd Q_2)\,)\end{aligned}$$

A sequence of designs $(P_1 \vdash Q_1)$ and $(P_2 \vdash Q_2)$ terminates when P_1 holds, and Q_1 is guaranteed to establish P_2. On termination, the sequence establishes the composition of the postconditions.

$$\begin{aligned}\textbf{\textit{T3}}\quad & (\,(P_1 \vdash Q_1)\,;\,(P_2 \vdash Q_2)\,)\\ & = (\,(\neg\,(\neg\,P_1\,;\,\textbf{\textit{true}}) \wedge (Q_1\,\textbf{\textit{wp}}\,P_2)) \vdash (Q_1\,;\,Q_2)\,)\end{aligned}$$

where $Q_1\,\textbf{\textit{wp}}\,P_2$ is the weakest precondition under which execution of Q_1 is guaranteed to achieve the postcondition P_2. It is defined in [15] as

$$Q\,\textbf{\textit{wp}}\,P = \neg\,(Q\,;\,\neg\,P)$$

The Isabelle proof of this fact is difficult, but rewarding:

Example 18 (Isar Proof of Design Composition).

theorem *DesignD-composition*:
assumes
  $(P1 \in \mathit{WF\text{-}RELATION})$ $(P2 \in \mathit{WF\text{-}RELATION})$
  $(Q1 \in \mathit{WF\text{-}RELATION})$ $(Q2 \in \mathit{WF\text{-}RELATION})$
shows ‘$(P1 \vdash Q1); (P2 \vdash Q2)$‘=‘$(\neg(\neg P1\ ;\ \mathit{true})) \wedge (Q1\ \mathit{wp}\ P2) \vdash (Q1; Q2)$‘
proof –
  have ‘$(P1 \vdash Q1)\ ;\ (P2 \vdash Q2)$‘
    = ‘$\exists\ \mathit{okay}'''\ .\ ((P1 \vdash Q1)[\$\mathit{okay}'''/\mathit{okay}']\ ;\ (P2 \vdash Q2)[\$\mathit{okay}'''/\mathit{okay}])$‘
   by (*smt DesignD-rel-closure MkPlain-UNDASHED SemiR-extract-variable assms*)

  also have ... = ‘ $((P1 \vdash Q1)[\mathit{false}/\mathit{okay}']\ ;\ (P2 \vdash Q2)[\mathit{false}/\mathit{okay}])$
          $\vee\ ((P1 \vdash Q1)[\mathit{true}/\mathit{okay}']\ ;\ (P2 \vdash Q2)[\mathit{true}/\mathit{okay}])$‘
   by (*simp add*:*ucases typing usubst defined closure unrest DesignD-def assms*)

  also have ... = ‘$((\mathit{ok} \wedge P1 \Rightarrow Q1)\ ;\ (P2 \Rightarrow \mathit{ok}' \wedge Q2)) \vee ((\neg\ (\mathit{ok} \wedge P1))\ ;\ \mathit{true})$‘
   by (*simp add*: *typing usubst defined unrest DesignD-def OrP-comm assms*)

also have ... = ‘$((\neg\ (ok \wedge P1)\ ;\ (P2 \Rightarrow ok' \wedge Q2)) \vee \neg\ (ok \wedge P1)\ ;\ true)$
$\vee\ Q1\ ;\ (P2 \Rightarrow ok' \wedge Q2)$‘
by (*smt OrP-assoc OrP-comm SemiR-OrP-distr ImpliesP-def*)

also have ... = ‘$(\neg\ (ok \wedge P1)\ ;\ true) \vee Q1\ ;\ (P2 \Rightarrow ok' \wedge Q2)$‘
by (*smt SemiR-OrP-distl utp-pred-simps*(*9*))

also have ... = ‘$(\neg ok\ ;\ true) \vee (\neg P1\ ;\ true) \vee (Q1\ ;\ \neg P2) \vee (ok' \wedge (Q1\ ;\ Q2))$‘
proof –
from *assms* **have** ‘$Q1\ ;\ (P2 \Rightarrow ok' \wedge Q2)$‘ = ‘$(Q1\ ;\ \neg P2) \vee (ok' \wedge (Q1\ ;\ Q2))$‘
by (*smt AndP-comm SemiR-AndP-right-postcond ImpliesP-def SemiR-OrP-distl*)

thus *?thesis* **by** (*smt OrP-assoc SemiR-OrP-distr demorgan2*)
qed

also have ... = ‘$(\neg\ (\neg\ P1\ ;\ true) \wedge \neg\ (Q1\ ;\ \neg\ P2)) \vdash (Q1\ ;\ Q2)$‘
proof –
have ‘$(\neg\ ok)\ ;\ true \vee (\neg\ P1)\ ;\ true$‘ = ‘$\neg\ ok \vee (\neg\ P1)\ ;\ true$‘
by (*simp add*: *SemiR-TrueP-precond closure*)

thus *?thesis*
by (*smt DesignD-def ImpliesP-def OrP-assoc demorgan2 demorgan3*)
qed

finally show *?thesis* **by** (*simp add*: *WeakPrecondP-def*)
qed

Preconditions can be relations, and this fact complicates the statement of Law ***T3***; if the P_1 is a condition instead, then the law is simplified as follows.

T3′ $((p_1 \vdash Q_1)\ ;\ (P_2 \vdash Q_2)) = (p_1 \wedge (Q_1\ \textbf{\textit{wp}}\ P_2)) \vdash (Q_1\ ;\ Q_2))$

Example 19 (Simplifying condition-composition).

$\neg\ (\neg\ p_1\ ;\ \textbf{\textit{true}})$	*composition*
$= \neg\ \exists\, v_0 \bullet \neg\ p_1[v_0/v'] \wedge \textbf{\textit{true}}[v_0/v]$	v *not free in* ***true***
$= \neg\ \exists\, v_0 \bullet \neg\ p_1[v_0/v'] \wedge \textbf{\textit{true}}$	*unit for conjunction*
$= \neg\ \exists\, v_0 \bullet \neg\ p_1[v_0/v']$	v' *not free in* p_1
$= \neg\ \exists\, v_0 \bullet \neg\ p_1$	v_0 *not free in* p_1
$= \neg\ \neg\ p_1$	*propositional calculus*
$= p_1$	

□

A recursively defined design has as its body a function on designs; as such, it can be seen as a function on precondition-postcondition pairs (X, Y). Moreover, since the result of the function is itself a design, it can be written in terms of a pair of functions F and G, one for the precondition and one for the postcondition.

As the recursive design is executed, the precondition F is required to hold over and over again. The strongest recursive precondition so obtained has to be satisfied, if we are to guarantee that the recursion terminates. Similarly, the postcondition is established over and over again, in the context of the precondition. The weakest result that can possibly be obtained is that which can be guaranteed by the recursion.

$$\textbf{\textit{T4}} \quad (\,\mu\, X, Y \bullet (\,F(X, Y) \vdash G(X, Y)\,)\,) = (\,P(Q) \vdash Q\,)$$

$$\textbf{\textit{where}}\ P(Y) = (\,\nu X \bullet F(X, Y)\,)\ \textbf{\textit{and}}\ Q = (\,\mu\, Y \bullet P(Y) \Rightarrow G(P(Y), Y)\,)$$

Further intuition comes from the realisation that we want the least refined fixed-point of the pair of functions. That comes from taking the strongest precondition, since the precondition of every refinement must be weaker, and the weakest postcondition, since the postcondition of every refinement must be stronger.

10 Healthiness Conditions

Another way of characterising the set of designs is by imposing healthiness conditions on the alphabetised predicates. Hoare & He [15] identify four healthiness conditions that they consider of interest: ***H1*** to ***H4***. We discuss each of them.

10.1 *H1*: Unpredictability

A relation R is ***H1*** healthy if and only if $R = (ok \Rightarrow R)$. This means that observations cannot be made before the program has started. This healthiness condition is idempotent.

Law 101 (*H1* idempotent)

$$\textbf{\textit{H1}} \circ \textbf{\textit{H1}} = \textbf{\textit{H1}}$$

Proof:

$$\begin{array}{ll}
\quad \textbf{\textit{H1}} \circ \textbf{\textit{H1}}(P) & \textbf{\textit{H1}} \\
= ok \Rightarrow (ok \Rightarrow P) & \textit{propositional calculus} \\
= ok \wedge ok \Rightarrow P & \textit{propositional calculus} \\
= ok \Rightarrow P & \textbf{\textit{H1}} \\
= \textbf{\textit{H1}}(P) & \square
\end{array}$$

*Example 20 (Examples of **H1** relations).*

1. Abort, the bottom of the lattice, is healthy.

$$\mathbf{H1}(\mathbf{true}) \;=\; (ok \Rightarrow \mathbf{true}) \;=\; \mathbf{true}$$

2. Miracle, the top of the lattice, is healthy.

$$\mathbf{H1}(\neg\, ok) \;=\; (ok \Rightarrow \neg\, ok) \;=\; \neg\, ok$$

3. The following relation is healthy: $(ok \wedge x \neq 0 \Rightarrow x' < x)$.

$$\begin{aligned} &\mathbf{H1}(ok \wedge x \neq 0 \Rightarrow x' < x) \\ &= ok \Rightarrow (ok \wedge x \neq 0 \Rightarrow x' < x) \\ &= (ok \wedge x \neq 0 \Rightarrow x' < x) \end{aligned}$$

4. The following design is healthy: $(x \neq 0 \vdash x' < x)$.

$$\mathbf{H1}(x \neq 0 \vdash x' < x) \;=\; ok \Rightarrow (x \neq 0 \vdash x' < x) \;=\; (x \neq 0 \vdash x' < x)$$

□

If R is **H1**-healthy, then R also satisfies the left-zero and unit laws below.

$$\mathbf{true} \,;\, R = \mathbf{true} \quad \text{and} \quad I\!I_{\mathbf{D}} \,;\, R = R$$

We now present a proof of these results. First, we prove that the algebraic unit and zero properties guarantee **H1**-healthiness.

*Designs with Left-Units and Left-Zeros Are **H1**.*

$$\begin{aligned} & R && \textsf{assumption } (I\!I_{\mathbf{D}} \textsf{ is left-unit}) \\ &= I\!I_{\mathbf{D}} \,;\, R && I\!I_{\mathbf{D}} \textsf{ definition} \\ &= (\,\mathbf{true} \vdash I\!I_{\mathbf{D}}\,) \,;\, R && \textsf{design definition} \\ &= (\,ok \Rightarrow ok' \wedge I\!I\,) \,;\, R && \textsf{relational calculus} \\ &= (\neg\, ok \,;\, R\,) \vee (\,I\!I \,;\, R\,) && \textsf{relational calculus} \\ &= (\neg\, ok \,;\, \mathbf{true} \,;\, R\,) \vee (\,I\!I \,;\, R\,) && \textsf{assumption } (\mathbf{true} \textsf{ is left-zero}) \\ &= \neg\, ok \vee (\,I\!I \,;\, R\,) && \textsf{assumption } (I\!I \textsf{ is left-unit}) \\ &= \neg\, ok \vee R && \textsf{relational calculus} \\ &= ok \Rightarrow R && \square \end{aligned}$$

The Isabelle proof has a few more steps, but follows a similar line of reasoning. We require that P be a well-formed relation, consisting of only undashed and dashed variables. We also prefer the use of the simplifier, executed by simp, to discharge each of the steps.

```
theorem H1-algebraic-intro:
  assumes
    R ∈ WF-RELATION
    (true ;  R = true)
    (II_D ;  R = R)
  shows R is H1
proof –
  let ?vs = REL-VAR – {okay,okay´}
  have R = II_D ;  R by (simp add: assms)
  also have ... = ‘(true ⊢ II ?vs) ;  R‘
    by (simp add:SkipD-def)
  also have ... = ‘(ok ⇒ (ok′ ∧ II ?vs)) ;  R‘
    by (simp add:DesignD-def)
  also have ... = ‘(ok ⇒ (ok ∧ ok′ ∧ II ?vs)) ;  R‘
    by (smt ImpliesP-export)
  also have ... = ‘(ok ⇒ (ok ∧ $okay´ = $okay ∧ II ?vs)) ;  R‘
    by (simp add: VarP-EqualP-aux typing defined, utp-rel-auto-tac)
  also have ... = ‘(ok ⇒ II) ;  R‘
    by (simp add:SkipRA-unfold[THEN sym]
        SkipR-as-SkipRA ImpliesP-export[THEN sym])
  also have ... = ‘((¬ ok) ;  R ∨ R)‘
    by (simp add:ImpliesP-def SemiR-OrP-distr)
  also have ... = ‘(((¬ ok) ;  true) ;  R ∨ R)‘
    by (simp add:SemiR-TrueP-precond closure)
  also have ... = ‘((¬ ok) ;  true ∨ R)‘
    by (simp add:SemiR-assoc[THEN sym] assms)
  also have ... = ‘ok ⇒ R‘
    by (simp add:SemiR-TrueP-precond closure ImpliesP-def)
  finally show ?thesis by (simp add:is-healthy-def H1-def)
qed
```

Next, we prove the implication the other way around: that ***H1***-healthy predicates have the unit and zero properties.

H1 *Predicates Have a Left-Zero.*

$$
\begin{array}{ll}
\ \textbf{\textit{true}} \,;\, R & \textit{assumption } (R \textit{ is } \textbf{\textit{H1}}) \\
= \textbf{\textit{true}} \,;\, (\, ok \Rightarrow R \,) & \textit{relational calculus} \\
= (\, \textbf{\textit{true}} \,;\, \neg\, ok \,) \vee (\, \textbf{\textit{true}} \,;\, R \,) & \textit{relational calculus} \\
= \textbf{\textit{true}} \vee (\, \textbf{\textit{true}} \,;\, R \,) & \textit{relational calculus} \\
= \textbf{\textit{true}} & \Box
\end{array}
$$

. . . and the same in Isabelle:

theorem *H1-left-zero*:
  assumes
    P ∈ *WF-RELATION*
    P is H1
  shows *true* ; *P* = *true*
proof −
  from *assms* **have** '*true* ; *P*' = '*true* ; (*ok* ⇒ *P*)'
    by (*simp add*:*is-healthy-def H1-def*)
  also have ... = '*true* ; (¬ *ok* ∨ *P*)'
    by (*simp add*:*ImpliesP-def*)
  also have ... = '(*true* ; ¬ *ok*) ∨ (*true* ; *P*)'
    by (*simp add*:*SemiR-OrP-distl*)
  also from *assms* **have** ... = '*true* ∨ (*true* ; *P*)'
    by (*simp add*:*SemiR-precond-left-zero closure*)
  finally show *?thesis* **by** *simp*
qed

H1 *Predicates Have a Left-Unit.*

$\mathit{II}_{\boldsymbol{D}} \; ; R$	*definition of* $\mathit{II}_{\boldsymbol{D}}$
$= (\, \textbf{\textit{true}} \vdash \mathit{II}_{\boldsymbol{D}} \,) \; ; R$	*definition of design*
$= (\, ok \Rightarrow ok' \land \mathit{II} \,) \; ; R$	*relational calculus*
$= (\, \neg\, ok \; ; R \,) \lor (\, ok \land R \,)$	*relational calculus*
$= (\, \neg\, ok \; ; \textbf{\textit{true}} \; ; R \,) \lor (\, ok \land R \,)$	***true*** *is left-zero*
$= (\, \neg\, ok \; ; \textbf{\textit{true}} \,) \lor (\, ok \land R \,)$	*relational calculus*
$= \neg\, ok \lor (\, ok \land R \,)$	*relational calculus*
$= ok \Rightarrow R$	R *is* ***H1***
$= R$	□

. . . and the same in Isabelle:

theorem *H1-left-unit*:
  assumes
    P ∈ *WF-RELATION*
    P is H1
  shows II_D ; *P* = *P*
proof −
  let *?vs* = *REL-VAR* − {*okay*,*okay´*}
  have II_D ; *P* = '(*true* ⊢ $II_{?vs}$) ; *P*' **by** (*simp add*:*SkipD-def*)
  also have ... = '(*ok* ⇒ *ok'* ∧ $II_{?vs}$) ; *P*'
    by (*simp add*:*DesignD-def*)
  also have ... = '(*ok* ⇒ *ok* ∧ *ok'* ∧ $II_{?vs}$) ; *P*'
    by (*smt ImpliesP-export*)
  also have ... = '(*ok* ⇒ *ok* ∧ \$*okay´* = \$*okay* ∧ $II_{?vs}$) ; *P*'
    by (*simp add*: *VarP-EqualP-aux*, *utp-rel-auto-tac*)
  also have ... = '(*ok* ⇒ *II*) ; *P*'
    by (*simp add*: *SkipRA-unfold*[*of okay*] *ImpliesP-export*[*THEN sym*])
  also have ... = '((¬ *ok*) ; *P* ∨ *P*)'

```
      by (simp add:ImpliesP-def SemiR-OrP-distr)
    also have ... = ‘(((¬ ok) ; true) ; P ∨ P)‘
      by (metis NotP-cond-closure SemiR-TrueP-precond VarP-cond-closure)
    also have ... = ‘((¬ ok) ; (true ; P) ∨ P)‘
      by (metis SemiR-assoc)
    also from assms have ... = ‘(ok ⇒ P)‘
      by (simp add:H1-left-zero ImpliesP-def SemiR-TrueP-precond closure)
    finally show ?thesis using assms
      by (simp add:H1-def is-healthy-def)
  qed
```

This means that we can use the left-zero and unit laws to exactly characterise ***H1*** healthiness. We can assert this equivalence property in Isabelle by combining the three theorems:

```
  theorem H1-algebraic:
    assumes R ∈ WF-RELATION
    shows R is H1 ⟷ (true ; R = true) ∧ (II_D ; R = R)
     by (metis H1-algebraic-intro H1-left-unit H1-left-zero assms)
```

The design identity is the obvious lifting of the relational identity to a design; that is, it has precondition ***true*** and the postcondition is the relational identity. There's a simple relationship between them: ***H1***.

Law 102 (Relational and design identities)

$$\mathit{II}_{\mathbf{D}} \;=\; \mathbf{H1}(\mathit{II})$$

Proof:

$\mathit{II}_{\mathbf{D}}$	$\mathit{II}_{\mathbf{D}}$
$= (\mathbf{true} \vdash \mathit{II})$	*design*
$= (ok \Rightarrow ok' \wedge \mathit{II})$	II, *prop calculus*
$= (ok \Rightarrow \mathit{II})$	***H1***
$= \mathbf{H1}(\mathit{II})$	$\square$

```
theorem H1-algebraic-intro:
  assumes
    R ∈ WF-RELATION
    (true ; R = true)
    (II_D ; R = R)
  shows R is H1
proof –
  let ?vs = REL-VAR – {okay,okay´}
  have R = II_D ; R by (simp add: assms)
  also have ... = ‘(true ⊢ II_?vs) ; R‘
    by (simp add:SkipD-def)
  also have ... = ‘(ok ⇒ (ok´ ∧ II_?vs)) ; R‘
    by (simp add:DesignD-def)
  also have ... = ‘(ok ⇒ (ok ∧ ok´ ∧ II_?vs)) ; R‘
```

    by (*smt ImpliesP-export*)
  also have ... = ‘$(ok \Rightarrow (ok \wedge \$okay´ = \$okay \wedge II_{?vs}))$; R‘
    by (*simp add*: *VarP-EqualP-aux typing defined*, *utp-rel-auto-tac*)
  also have ... = ‘$(ok \Rightarrow II)$; R‘
    by (*simp add*:*SkipRA-unfold*[*THEN sym*]
        SkipR-as-SkipRA ImpliesP-export[*THEN sym*])
  also have ... = ‘$((\neg\ ok)$; $R \vee R)$‘
    by (*simp add*:*ImpliesP-def SemiR-OrP-distr*)
  also have ... = ‘$(((\neg\ ok)$; $true)$; $R \vee R)$‘
    by (*simp add*:*SemiR-TrueP-precond closure*)
  also have ... = ‘$((\neg\ ok)$; $true \vee R)$‘
    by (*simp add*:*SemiR-assoc*[*THEN sym*] *assms*)
  also have ... = ‘$ok \Rightarrow R$‘
    by (*simp add*:*SemiR-TrueP-precond closure ImpliesP-def*)
  finally show *?thesis* **by** (*simp add*:*is-healthy-def H1-def*)
qed

10.2 *H2*: Possible Termination

The second healthiness condition is $[\,R[false/ok'] \Rightarrow R[true/ok']\,]$. This means that if R is satisfied when ok' is *false*, it is also satisfied then ok' is *true*. In other words, R cannot *require* nontermination, so that it is always possible to terminate.

Example 21 (Example ***H2*** *predicates).*

1. $\bot_{\boldsymbol{D}}$

$$\bot_{\boldsymbol{D}}^f = \textbf{\textit{true}}^f = \textbf{\textit{true}} = \textbf{\textit{true}}^t = \bot_{\boldsymbol{D}}^t$$

2. $\top_{\boldsymbol{D}}$

$$\top^f = (\neg\ ok)^f = \neg\ ok = (\neg\ ok)^t = \top_{\boldsymbol{D}}^t$$

3. $(ok' \wedge (x' = 0))$

$$(ok' \wedge (x' = 0))^f = \textbf{\textit{false}} \Rightarrow (x' = 0) = (ok' \wedge x' = 0)^t$$

4. $(x \neq 0 \vdash x' < x)$

$$\begin{aligned}
&(x \neq 0 \vdash x' < x)^f \\
&= (ok \wedge x \neq 0 \Rightarrow ok' \wedge x' < x)^f \\
&= (ok \wedge x \neq 0 \Rightarrow \textbf{\textit{false}}) \\
&\Rightarrow (ok \wedge x \neq 0 \Rightarrow x' < x) \\
&= (ok \wedge x \neq 0 \Rightarrow ok' \wedge x' < x)^t \\
&= (x \neq 0 \vdash x' < x)^t
\end{aligned}$$

□

The healthiness condition ***H2*** is not obviously characterised by a monotonic idempotent function. We now define the idempotent J for alphabet $\{ok, ok', v, v'\}$, and use this in an alternative definition of ***H2***.

$$J \;\widehat{=}\; (ok \Rightarrow ok') \wedge v' = v$$

The most interesting property of J is the following algebraic law that allows a relation to be split into two complementary parts, one that definitely aborts and one that does not. Note the asymmetry between the two parts.

Law 103 (J-split) *For all relations with ok and ok′ in their alphabet,*

$$P \,;\, J \;=\; P^f \vee (P^t \wedge ok')$$

Proof:

$$\begin{array}{ll}
\quad P \,;\, J & \textit{J} \\
= P \,;\, (ok \Rightarrow ok') \wedge v' = v & \textit{propositional calculus} \\
= P \,;\, (ok \Rightarrow ok \wedge ok') \wedge v' = v & \textit{propositional calculus} \\
= P \,;\, (\neg\, ok \vee ok \wedge ok') \wedge v' = v & \textit{relational calculus} \\
= P \,;\, \neg\, ok \wedge v' = v & \textit{right one-point, twice} \\
\quad \vee & \\
\quad (P \,;\, ok \wedge v' = v) \wedge ok' & \\
= P^f \vee (P^t \wedge ok') & \Box
\end{array}$$

Likewise this proof can be mechanised in Isabelle, though a little more detailed is required. In particular, we treat the equalities of each sides of the disjunction separately in the final step.

theorem *J-split*:
  assumes $P \in$ *WF-RELATION*
  shows ‘P ; J‘ = ‘$P^f \vee (P^t \wedge ok')$‘
proof –
  let *?vs* = (*REL-VAR* – {*okay*,*okay*´})

  have ‘P ; J‘ = ‘P ; $((ok \Rightarrow ok') \wedge II_{?vs})$‘ **by** (*simp add*:*J-pred-def*)

  also have ... = ‘P ; $((ok \Rightarrow ok \wedge ok') \wedge II_{?vs})$‘ **by** (*smt ImpliesP-export*)

  also have ... = ‘P ; $((\neg\, ok \vee (ok \wedge ok')) \wedge II_{?vs})$‘ **by** (*utp-rel-auto-tac*)

  also have ... = ‘$(P$; $(\neg\, ok \wedge II_{?vs})) \vee (P$; $(ok \wedge (II_{?vs} \wedge ok')))$‘
   by (*smt AndP-OrP-distr AndP-assoc AndP-comm SemiR-OrP-distl*)

  also have ... = ‘$P^f \vee (P^t \wedge ok')$‘
  proof –
   from *assms* **have** ‘$(P$; $(\neg\, ok \wedge II_{?vs}))$‘ = ‘$P^f$‘
    by (*simp add*: *SemiR-left-one-point SkipRA-right-unit*)

```
    moreover have ‘(P ; (ok ∧ II ?vs ∧ ok′))‘ = ‘(P^t ∧ ok′)‘
    proof –
      from assms have ‘(P ; (ok ∧ II ?vs ∧ ok′))‘ = ‘(P ; (ok ∧ II ?vs)) ∧ ok′‘
        by (utp-xrel-auto-tac)

      moreover from assms have ‘(P ; (ok ∧ II ?vs))‘ = ‘P^t‘
        by (simp add: SemiR-left-one-point SkipRA-right-unit )

      finally show ?thesis .
    qed

    ultimately show ?thesis by simp
  qed

  finally show ?thesis .
qed
```

The two characterisations of ***H2*** are equivalent.

Law 104 (*H2* equivalence)

$$(P = P \, ; J) \;=\; [\, P^f \Rightarrow P^t \,]$$

Proof:

$$\begin{array}{ll}
\quad (P = P \, ; J) & \textit{J-split} \\
= (P = P^f \vee (P^t \wedge ok')) & \textit{ok}' \textit{ split} \\
= (P = P^f \vee (P^t \wedge ok'))^f \wedge (P = P^f \vee (P^t \wedge ok'))^t & \textit{subst.} \\
= (P^f = P^f \vee (P^t \wedge \textbf{\textit{false}})) \wedge (P^t = P^f \vee (P^t \wedge \textbf{\textit{true}})) & \textit{prop calc.} \\
= (P^f = P^f) \wedge (P^t = P^f \vee P^t) & \textit{reflection} \\
= (P^t = P^f \vee P^t) & \textit{predicate calculus} \\
= [\, P^f \Rightarrow P^t \,] & \square
\end{array}$$

. . . and in Isabelle:

```
theorem H2-equivalence:
  assumes R ∈ WF-RELATION
  shows R is H2 ⟷ [R^f ⇒ R^t]
proof –
  from assms have ‘[R ⇔ (R ; J)]‘ = ‘[R ⇔ (R^f ∨ (R^t ∧ ok′))]‘
    by (simp add:J-split)

  also have ... = ‘[(R ⇔ R^f ∨ R^t ∧ ok′)^f ∧ (R ⇔ R^f ∨ R^t ∧ ok′)^t]‘
    by (simp add:ucases)

  also have ... = ‘[(R^f ⇔ R^f) ∧ (R^t ⇔ R^f ∨ R^t)]‘
    by (simp add:usubst closure typing defined)
  also have ... = ‘[$R^t \Leftrightarrow (R^f \vee R^t)$]‘
    by (utp-pred-tac)

  finally show ?thesis
    by (utp-pred-auto-tac)
qed
```

J itself is ***H2*** healthy.

Law 105 (J is *H2*)

$$J = \boldsymbol{H2}(J)$$

Proof:

$\boldsymbol{H2}(J)$	*J-split*
$= J^f \vee (J^t \wedge ok')$	*J*
$= (\neg\, ok \wedge v' = v) \vee (ok' \wedge v' = v)$	*propositional calculus*
$= (\neg\, ok \vee ok') \wedge v' = v$	*propositional calculus*
$= (ok \Rightarrow ok') \wedge v' = v$	*J*
$= J$	$\square$

. . . and in Isabelle:

```
theorem J-is-H2:
  H2(J) = J
proof -
  let ?vs = (REL-VAR - {okay,okay´})
  have H2(J) = ‘$J^f \vee (J^t \wedge ok')$‘
    by (metis H2-def J-closure J-split)

  also have ... = ‘$((\neg\, ok \wedge II_{?vs}) \vee II_{?vs} \wedge ok')$‘
    by (simp add:J-pred-def usubst typing defined closure)

  also have ... = ‘$(\neg\, ok \vee ok') \wedge II_{?vs}$‘
    by (utp-pred-auto-tac)

  also have ... = ‘$(ok \Rightarrow ok') \wedge II_{?vs}$‘
    by (utp-pred-tac)

  finally show ?thesis
    by (metis J-pred-def)
qed
```

J is idempotent.

Law 106 (*H2*-idempotent)

$$\mathbf{H2} \circ \mathbf{H2} \;=\; \mathbf{H2}$$

Proof:

$$\begin{aligned}
& \mathbf{H2} \circ \mathbf{H2}(P) && \mathbf{H2} \\
&= (P \,;\, J) \,;\, J && \textit{associativity} \\
&= P \,;\, (J \,;\, J) && \mathbf{H2} \\
&= P \,;\, \mathbf{H2}(J) && J\ \mathbf{H2}\ \textit{healthy} \\
&= P \,;\, J && \mathbf{H2} \\
&= P && \square
\end{aligned}$$

. . . and in Isabelle:

```
theorem H2-idempotent:
  ‘H2 (H2 R)‘ = ‘H2 R‘
proof −
  have ‘H2 (H2 R)‘ = ‘(R ; J) ; J‘
    by (metis H2-def)
  also have ... = ‘R ; (J ; J)‘
    by (metis SemiR-assoc)
  also have ... = ‘R ; H2 J‘
    by (metis H2-def)
  also have ... = ‘R ; J‘
    by (metis J-is-H2)
  also have ... = ‘H2 R‘
    by (metis H2-def)
  finally show ?thesis .
qed
```

Any predicate that insists on proper termination is healthy.

*Example 22 (Example: **H2**-substitution).*

$ok' \wedge (x' = 0)$ is ***H2***

Proof:

$$\begin{aligned}
& (ok' \wedge (x' = 0))^f \Rightarrow (ok' \wedge (x' = 0))^t \\
&= (\mathbf{false} \wedge (x' = 0) \Rightarrow \mathbf{true} \wedge (x' = 0)) \\
&= (\mathbf{false} \Rightarrow (x' = 0)) \\
&= \mathbf{true}
\end{aligned}$$

$\square$

The proof could equally well be done with the alternative characterisation of ***H2***.

Example 23. Example: ***H2***-*J*

$ok' \wedge (x' = 0)$ is ***H2***

Proof:

$$\begin{aligned}
& ok' \wedge (x' = 0) \ ; J && \textsf{\textit{J-splitting}} \\
= \ & (ok' \wedge (x' = 0))^f \vee ((ok' \wedge (x' = 0))^t \wedge ok') && \textsf{\textit{subst.}} \\
= \ & (\textbf{\textsf{\textit{false}}} \wedge (x' = 0)) \vee (\textbf{\textsf{\textit{true}}} \wedge (x' = 0) \wedge ok') && \textsf{\textit{prop. calculus}} \\
= \ & \textbf{\textsf{\textit{false}}} \vee ((x' = 0) \wedge ok') && \textsf{\textit{propositional calculus}} \\
= \ & ok' \wedge (x' = 0)
\end{aligned}$$

□

If a relation is both ***H1*** and ***H2*** healthy, then it is a design. We prove this by showing that the relation can be expressed syntactically as a design.

Law 107 (*H1-H2* relations are designs)

$$\begin{aligned}
& P && \textsf{\textit{assumption: P is \textbf{H1}}} \\
= \ & ok \Rightarrow P && \textsf{\textit{assumption: P is \textbf{H2}}} \\
= \ & ok \Rightarrow P \ ; \ J && \textsf{\textit{J-splitting}} \\
= \ & ok \Rightarrow P^f \vee (P^t \wedge ok') && \textsf{\textit{propositional calculus}} \\
= \ & ok \wedge \neg \ P^f \Rightarrow ok' \wedge P^t && \textsf{\textit{design}} \\
= \ & \neg \ P^f \vdash P^t && \square
\end{aligned}$$

Likewise this proof can be formalised in Isabelle:

```
theorem H1-H2-is-DesignD:
  assumes
    P ∈ WF-RELATION
    P is H1
    P is H2
  shows P = ‘(¬ P^f) ⊢ P^t‘
proof –
  have P = ‘ok ⇒ P‘
    by (metis H1-def assms(2) is-healthy-def)

  also have ... = ‘ok ⇒ (P ; J)‘
    by (metis H2-def assms(3) is-healthy-def)

  also have ... = ‘ok ⇒ (P^f ∨ (P^t ∧ ok'))‘
    by (metis J-split assms(1))

  also have ... = ‘ok ∧ (¬ P^f) ⇒ ok' ∧ P^t‘
    by (utp-pred-auto-tac)
  also have ... = ‘¬ P^f ⊢ P^t ‘
    by (metis DesignD-def)

  finally show ?thesis .
qed
```

Designs are obviously ***H1***; we now show that they must also be ***H2***. These two results complete the proof that ***H1*** and ***H2*** together exactly characterise designs.

Law 108 *Designs are* ***H2***

$$\begin{array}{ll}
(\,P \vdash Q\,)^f & \textit{definition of design}\\
= (\,ok \wedge P \Rightarrow \textbf{\textit{false}}) & \textit{propositional calculus}\\
\Rightarrow (\,ok \wedge P \Rightarrow Q\,) & \textit{definition of design}\\
= (\,P \vdash Q\,)^t & \square
\end{array}$$

Miracle, even though it does not mention ok', is ***H2***-healthy.

Example 24 (Miracle is ***H2****).*

$$\begin{array}{ll}
\neg\ ok & \textit{miracle}\\
= \textbf{\textit{true}} \vdash \textbf{\textit{false}} & \textit{designs are } \textbf{\textit{H2}}\\
= \textbf{\textit{H2}}(\textbf{\textit{true}} \vdash \textbf{\textit{false}}) & \textit{miracle}\\
= \textbf{\textit{H2}}(\neg\ ok) &
\end{array}$$

$\square$

The final thing to prove is that it does not matter in which order we apply ***H1*** and ***H2***; the key point is that a design requires both properties.

Law 109 (*H1*-*H2* commute)

$$\begin{array}{ll}
\textbf{\textit{H1}} \circ \textbf{\textit{H2}}(P) & \textbf{\textit{H1}},\ \textbf{\textit{H2}}\\
= ok \Rightarrow P \,;\, J & \textit{propositional calculus}\\
= \neg\ ok\ \vee\ P \,;\, J & \textit{miracle is } \textbf{\textit{H2}}\\
= \textbf{\textit{H2}}(\neg\ ok)\ \vee\ P \,;\, J & \textbf{\textit{H2}}\\
= \neg\ ok \,;\, J\ \vee\ P \,;\, J & \textit{relational calculus}\\
= (\neg\ ok \vee P) \,;\, J & \textit{propositional calculus}\\
= (ok \Rightarrow P) \,;\, J & \textbf{\textit{H1}},\ \textbf{\textit{H2}}\\
= \textbf{\textit{H2}} \circ \textbf{\textit{H1}}(P) & \square
\end{array}$$

10.3 *H3*: Dischargeable Assumptions

The healthiness condition ***H3*** is specified as an algebraic law: $R = R \,;\, \mathit{II}_{\mathbf{D}}$. A design satisfies ***H3*** exactly when its precondition is a condition. This is a very desirable property, since restrictions imposed on dashed variables in a precondition can never be discharged by previous or successive components. For example, $x' = 2 \vdash true$ is a design that can either terminate and give an arbitrary value to x, or it can give the value 2 to x, in which case it is not required to terminate. This is a rather bizarre behaviour.

A Design Is ***H3 iff*** *Its Assumption Is a Condition.*

$$
\begin{array}{ll}
\quad ((P \vdash Q) = ((P \vdash Q) ; I\!I_{\mathbf{D}})) & \textit{definition of design-skip} \\
= ((P \vdash Q) = ((P \vdash Q) ; (\mathbf{true} \vdash I\!I_{\mathbf{D}}))) & \textit{sequence of designs} \\
= ((P \vdash Q) = (\neg (\neg P ; \mathbf{true}) \land \neg (Q ; \neg \mathbf{true}) \vdash Q ; I\!I_{\mathbf{D}})) & \textit{skip unit} \\
= ((P \vdash Q) = (\neg (\neg P ; \mathbf{true}) \vdash Q)) & \textit{design equality} \\
= (\neg P = \neg P ; \mathbf{true}) & \textit{propositional calculus} \\
= (P = P ; \mathbf{true}) & \square
\end{array}
$$

The final line of this proof states that $P = \exists v' \bullet P$, where v' is the output alphabet of P. Thus, none of the after-variables' values are relevant: P is a condition only on the before-variables.

10.4 *H4*: Feasibility

The final healthiness condition is also algebraic: $R ; \mathbf{true} = \mathbf{true}$. Using the definition of sequence, we can establish that this is equivalent to $\exists v' \bullet R$, where v' is the output alphabet of R. In words, this means that for *every* initial value of the observational variables on the input alphabet, there exist final values for the variables of the output alphabet: more concisely, establishing a final state is feasible. The design $\top_{\mathbf{D}}$ is not ***H4*** healthy, since miracles are not feasible.

11 Related Work

Our mechanisation of UTP theories of relations and of designs and our future mechanisation of the theory of reactive processes form the basis for reasoning about a family of modern multi-paradigm modelling languages. This family contains both *Circus* [34,35] and CML [37]: *Circus* combines Z [28] and CSP [14], whilst CML combines VDM [16] and CSP. Both languages are based firmly on the notion of refinement [25] and have a variety of extensions with additional computational paradigms, including real-time [27,31], object orientation [8], synchronicity [5], and process mobility [29,30]. Further information on *Circus* may be found at `www.cs.york.ac.uk/circus`. CML is being developed as part of the European Framework 7 COMPASS project on Comprehensive Modelling for Advanced Systems of Systems (grant Agreement: 287829). See `www.compass-research.eu`.

Our implementation of UTP in Isabelle is a natural extension of the work in Oliveira's PhD thesis [20], which is extended in [22], where UTP is embedded in

ProofPowerZ, an extension of ProofPower/HOL supporting the Z notation, to mechanise the definition of *Circus*.

Feliachi et al. [10] have developed a machine-checked, formal semantics based on a shallow embedding of *Circus* in Isabelle/Circus, a semantic theory of UTP also based on Isabelle/HOL. The definitions of *Circus* are based on those in [21], which are in turn based on those in [35]. Feliachi et al. derive proof rules from this semantics and implement tactic support for proofs of refinement for *Circus* processes involving both data and behavioral aspects. This proof environment supports a syntax for the semantic definitions that is close to textbook presentations of Circus.

Our work differs from that of Feliachi et al. in three principle ways:

1. **Alphabets.** We have a unified type for predicates where alphabets are represented explicitly. Predicates with different alphabets can be composed without the need for type-level coercions, and variables can be easily added and removed.
2. **Expressivity.** Our encoding of predicates is highly flexible, providing support for many different operators and subtheories, of which binary relations is only one. We also can support complex operators on variables, in particular we provide a unified notion of substitution. The user can also supply their own type system for values, meaning we are not necessarily limited to the type-system of HOL.
3. **Meta-theoretic Proofs.** We give a deeper semantics to operators such as sequential composition and the quantifiers, rather than identifying them with operators of HOL, and therefore support proofs about the operators of the UTP operators. This meta-level reasoning allows us to perform soundness proofs about the denotational semantics of our language.

12 Conclusion

We have mechanised UTP, including alphabetised predicates, relations, the operators of imperative programming, and the theory of designs. All the proofs contained in this paper have been mechanised within our library, including those where the proofs have been omitted. Thus far, we have mechanised over 200 laws about the basic operators and over 40 laws about the theory of designs. The UTP library can therefore be used to perform basic static analysis of imperative programs. The proof automation level is high due to the inclusion of our proof tactics and the power of sledgehammer combined with the algebraic laws of the UTP.

There are several directions for future work:

– Mechanise additional theories, for instance CSP and *Circus*, which will give the ability to reason about reactive programs with concurrency.
– Complete the VDM/CML model, which includes implementing all the standard VDM library functions, so we can support verification of VDM specifications.

– Implement *proof obligations* for UTP expressions, so that ***H4*** healthiness of a program can be verified.
– Implement Z-schema types, which allow the specification of complex data structures.
– Implement the complete refinement calculus by mechanising refinement laws in Isabelle/UTP. This will allow the derivation of programs from high-level specifications, supported by mechanised proof.

All of this is leading towards proof support for CML, so that we can prove the validity of complex specifications of systems of systems in the COMPASS tool.

References

1. Abrial, J.-R.: The B-Book: Assigning Progams to Meanings. Cambridge University Press (1996)
2. Back, R.J.R., Wright, J.: Refinement Calculus: A Systematic Introduction. Graduate Texts in Computer Science. Springer (1998)
3. Blanchette, J.C., Nipkow, T.: Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 131–146. Springer, Heidelberg (2010)
4. Blanchette, J.C., Bulwahn, L., Nipkow, T.: Automatic Proof and Disproof in Isabelle/HOL. In: Tinelli, C., Sofronie-Stokkermans, V. (eds.) FroCoS 2011. LNCS, vol. 6989, pp. 12–27. Springer, Heidelberg (2011) §
5. Butterfield, A., Sherif, A., Woodcock, J.: Slotted-Circus. In: Davies, J., Gibbons, J. (eds.) IFM 2007. LNCS, vol. 4591, pp. 75–97. Springer, Heidelberg (2007)
6. Beg, A., Butterfield, A.: Linking a state-rich process algebra to a state-free algebra to verify software/hardware implementation. In: Proceedings of the 8th International Conference on Frontiers of Information Technology, FIT 2010, article 47 (2010)
7. Cavalcanti, A., Woodcock, J.: A tutorial introduction to CSP in Unifying Theories of Programming. In: Cavalcanti, A., Sampaio, A., Woodcock, J. (eds.) PSSE 2004. LNCS, vol. 3167, pp. 220–268. Springer, Heidelberg (2006)
8. Cavalcanti, A., Sampaio, A., Woodcock, J.: Unifying classes and processes. Software and System Modeling 4(3), 277–296 (2005)
9. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
10. Feliachi, A., Gaudel, M.-C., Wolff, B.: Isabelle/Circus: A Process Specification and Verification Environment. In: Joshi, R., Müller, P., Podelski, A. (eds.) VSTTE 2012. LNCS, vol. 7152, pp. 243–260. Springer, Heidelberg (2012)
11. Harwood, W., Cavalcanti, A., Woodcock, J.: A theory of pointers for the UTP. In: Fitzgerald, J.S., Haxthausen, A.E., Yenigun, H. (eds.) ICTAC 2008. LNCS, vol. 5160, pp. 141–155. Springer, Heidelberg (2008)
12. Hehner, E.C.R.: Retrospective and prospective for Unifying Theories of Programming. In: Dunne, S., Stoddart, B. (eds.) UTP 2006. LNCS, vol. 4010, pp. 1–17. Springer, Heidelberg (2006)
13. Hillenbrand, T., Buch, A., Vogt, R., Löchner, B.: Waldmeister: High-performance equational deduction. Journal of Automated Reasoning 18(2), 265–270 (1997)

14. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall (1985)
15. Hoare, C.A.R., He, J.: Unifying Theories of Programming. Series in Computer Science. Prentice Hall (1998)
16. Jones, C.B.: Systematic Software Development Using VDM. Prentice-Hall International (1986)
17. Morgan, C.: Programming from Specifications, 2nd edn. Prentice-Hall (1994)
18. Morris, J.M.: A Theoretical Basis for Stepwise Refinement and the Programming Calculus. Science of Computer Programming 9(3), 287–306 (1987)
19. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002)
20. Oliveira, M.V.M.: Formal derivation of state-rich reactive programs using Circus. PhD Thesis, Department of Computer Science, University of York, Report YCST-2006/02 (2005)
21. Oliveira, M., Cavalcanti, A., Woodcock, J.: A UTP semantics for Circus. Formal Asp. Comput. 21(1-2), 3–32 (2009)
22. Oliveira, M., Cavalcanti, A., Woodcock, J.: Unifying theories in ProofPower-Z. Formal Aspects of Computing 25(1), 133–158 (2013)
23. Perna, J.I., Woodcock, J.: UTP semantics for Handel-C. In: Butterfield, A. (ed.) UTP 2008. LNCS, vol. 5713, pp. 142–160. Springer, Heidelberg (2010)
24. Riazanov, A., Voronkov, A.: The design and implementation of Vampire. Journal of AI Communications 15(2/3), 91–110 (2002)
25. Sampaio, A., Woodcock, J., Cavalcanti, A.: Refinement in Circus. In: Eriksson, L.-H., Lindsay, P.A. (eds.) FME 2002. LNCS, vol. 2391, pp. 451–470. Springer, Heidelberg (2002)
26. Schultz, S.: E—A braniac theorem prover. Journal of AI Communications 15(2/3), 111–126 (2002)
27. Sherif, A., He, J.: Towards a Time Model for Circus. In: George, C.W., Miao, H. (eds.) ICFEM 2002. LNCS, vol. 2495, pp. 613–624. Springer, Heidelberg (2002)
28. Michael Spivey, J.: The Z Notation: A Reference Manual, 2nd edn. Series in Computer Science. Prentice Hall International (1992)
29. Tang, X., Woodcock, J.: Towards Mobile Processes in Unifying Theories. In: 2nd International Conference on Software Engineering and Formal Methods (SEFM 2004), Beijing, China, September 28–30, pp. 44–53. IEEE Computer Society (2004)
30. Tang, X., Woodcock, J.: Travelling Processes. In: Kozen, D., Shankland, C. (eds.) MPC 2004. LNCS, vol. 3125, pp. 381–399. Springer, Heidelberg (2004)
31. Wei, K., Woodcock, J., Cavalcanti, A.: *Circus Time* with Reactive Designs. In: Wolff, B., Gaudel, M.-C., Feliachi, A. (eds.) UTP 2012. LNCS, vol. 7681, pp. 68–87. Springer, Heidelberg (2013)
32. Weidenbach, C., Dimova, D., Fietzke, A., Kumar, R., Suda, M., Wischnewski, P.: SPASS Version 3.5. In: Schmidt, R.A. (ed.) CADE 2009. LNCS, vol. 5663, pp. 140–145. Springer, Heidelberg (2009)
33. Woodcock, J., Davies, J.: Using Z—Specification, Refinement, and Proof. Prentice-Hall (1996)
34. Woodcock, J., Cavalcanti, A.: A Concurrent Language for Refinement. In: Butterfield, A., Strong, G., Pahl, C. (eds.) 5th Irish Workshop on Formal Methods, IWFM 2001, Dublin, Ireland, July 16–17. BCS Workshops in Computing, pp. 16–17 (2001)
35. Woodcock, J., Cavalcanti, A.: The Semantics of *Circus*. In: Bert, D., Bowen, J.P., Henson, M.C., Robinson, K. (eds.) ZB 2002. LNCS, vol. 2272, pp. 184–203. Springer, Heidelberg (2002)

36. Woodcock, J., Cavalcanti, A.: A tutorial introduction to Designs in Unifying Theories of Programming. In: Boiten, E.A., Derrick, J., Smith, G. (eds.) IFM 2004. LNCS, vol. 2999, pp. 40–66. Springer, Heidelberg (2004)
37. Woodcock, J., Cavalcanti, A., Fitzgerald, J., Larsen, P., Miyazawa, A., Perry, S.: Features of CML: A formal modelling language for Systems of Systems. In: 7th IEEE International Conference on System of Systems Engineering (SoSE), pp. 1–6 (2012)
38. Zhan, N., Kang, E.Y., Liu, Z.: Component publications and compositions. In: Butterfield, A. (ed.) UTP 2008. LNCS, vol. 5713, pp. 238–257. Springer, Heidelberg (2010)
39. Zhu, H., Yang, F., He, J.: Generating denotational semantics from algebraic semantics for event-driven system-level language. In: Qin, S. (ed.) UTP 2010. LNCS, vol. 6445, pp. 286–308. Springer, Heidelberg (2010)

FORMULA 2.0: A Language for Formal Specifications

Ethan K. Jackson and Wolfram Schulte

Microsoft Research, Redmond, WA
{ejackson,schulte}@microsoft.com

Abstract. FORMULA 2.0 is a novel formal specification language based on *open-world logic programs* and *behavioral types*. Its goals are (1) succinct specifications of domain-specific abstractions and compilers, (2) efficient reasoning and compilation of input programs, (3) diverse synthesis and fast verification. We take a unique approach towards achieving these goals: Specifications are written as strongly-typed open-world logic programs. They are highly declarative and easily express rich synthesis / verification problems. Automated reasoning is enabled by efficient symbolic execution of logic programs into constraints. This tutorial introduces the FORMULA 2.0 language and concepts through a series of small examples.

1 Data and Types

1.1 Constants

FORMULA specifications define, examine, translate and generate *data*. The simplest kinds of data are *constants*, which have no internal structure. Every FORMULA specification has access to some predefined constants. *Numeric constants* can be (arbitrarily) long integers:

```
-1098245634534545630234, 0, 1, 2, 3098098445645649034
```

Or, they can be pairs of integers separated by a '.', i.e. decimal fractions:

```
-223423.23422342342, 0.0, 1.5, 10.8798723000000000000000003
```

Or, they can be pairs of integers separated by a '/', i.e. fractions:

```
-223423/23422342342, 4/8, 9873957/987395487987334
```

FORMULA converts numerics into normalized fractions; no precision is lost. For example, the following equalities are true:

```
2/3 = 6/9, 1/2 = 0.5, 1.0000 = 1
```

The following disequalities are true:

```
2/3 != 0.66667, 0 != 0.000000000000000000000000000001
```

Operations on numerics do not lose precision. Infinities are not explicitly part of FORMULA's vocabulary. For example, the fraction '1/0' causes a syntax error.

ASCII strings are also supported. One way to write a string is by enclosing it in double-quotes. These strings must end on the same line where they began, so we refer to them as *single-line strings*. Here are some examples:

```
"", "Hello World", "Foo\nBar"
```

Sometimes it is necessary to put special characters inside of a string. This can be accomplished using the C-style escape character '\'. Table 1 gives the escape sequences.

Some strings are unpleasant to escape, such as strings containing code or filenames with backslashes. Multi-line strings capture all the text within a pair of delimiters, including line breaks. A multi-line string starts with a single-double-quote pair `'"` and ends with a double-single-quote pair `"'`. Below is an example; note '\' is not an escape character in multi-line strings:

```
'" "This\string has funny 'thi\ngs in it'" "'
```

The only sequence that needs to be escaped in a multi-line string is the sequence terminating the string. For symmetry, the starting sequence also has an escape code (see Table 2). For example, the following equality is true:

```
'" ''"" ""'' \"' = " '\" \"' \\"
```

The final kind of constant is the *user-defined constant*. Syntactically, user-defined constants are identifiers. Here are some examples of user-defined constants:

```
TRUE, FALSE, RED, GREEN, NIL
```

Note that `TRUE` and `FALSE` are automatically defined on the user's behalf, though they are not keywords.

By convention, the names of user-defined constants should consist of all uppercase characters and be at least two characters long. This convention helps to distinguish constants from other identifiers. Two user-defined constants denote the same value are if and only if they have the same. For example:

```
TRUE = TRUE, FALSE = FALSE, TRUE != FALSE, RED != BLUE
```

1.2 Data Constructors

Complex data values are created by functions called *data constructors* (or *constructors* for short). An n-ary data constructor f takes n data values as arguments and returns a new data value. Here are some examples:

```
Person("John", "Smith"),
Node(1, Node(2, NIL, NIL), Node(3, NIL, NIL))
```

The $Person(,)$ constructor creates *Person* values from two string arguments. The $Node(,,)$ constructor builds binary trees of integer keys. The arguments to *Node* are: (1) an integer key, (2) a left-subtree (or the constant *NIL* if none) and (3) a right-subtree (or the constant *NIL* if none).

Data constructors are proper functions, i.e. they always produce the same values from the same inputs. Contrast this with the following object-oriented program:

```
class Node {
  ...
  Node (int Key, Node left, Node right)
  { ... }
}

Node x = new Node(1, null, null);
Node y = new Node(1, null, null);
if (x != y) {
   print("Different");
}
```

Table 1. Table of single-line string escapes

Single-Line String Escapes

Syntax	Result
`\n`	Produces a line feed.
`\r`	Produces a carriage return.
`\t`	Produces a tab.
`\`x	Produces x for $x \notin \{\mathtt{n}, \mathtt{r}, \mathtt{t}\}$, e.g. `\\` or `\"` .

Table 2. Table of multi-line string escapes

Multi-Line String Escapes

Syntax	Result
`''""`	Produces the sequence `'"`.
`""''`	Produces the sequence `"'`.

The program prints the string "Different" because x and y hold different nodes, even though the nodes were constructed with the same values. In FORMULA two values are the same if and only if (1) they are the same constant, or (2) they were constructed by the same constructor using the same arguments. For example:

```
NIL = NIL, NIL != FALSE,
Node(1, NIL, NIL) = Node(1, NIL, NIL),
Person("John", "Smith") != Node(2, NIL, NIL)
```

As this example shows, values can be compared using the *equality* '`=`' and *disequality* '`!=`' relations. These relations are defined for arbitrary pairs of values.

1.3 Ordering of Values

Values are also ordered. The ordering relation on values is called a *lexicographic order*, which generalizes the dictionary order of strings. First, we split all values into four ordered families: numerics, strings, user constants, and *complex values*:

Definition 11 (Ordering of Families)**.**

$$\mathit{family}(x) \overset{def}{=} \begin{cases} 0 & \text{if } x \text{ is a numeric,} \\ 1 & \text{if } x \text{ is a string,} \\ 2 & \text{if } x \text{ is a user constant,} \\ 3 & \text{otherwise.} \end{cases}$$

Families yield a *precedence relation* $\ll$ on values:

$$x \ll y \text{ if } \mathit{family}(x) < \mathit{family}(y).$$

Definition 12 (Ordering of Values)**.** For values x and y, define $x < y$ if $x \neq y$ and any of the following are satisfied:

– $x \ll y$.
– Both values are numerics; x comes before y on the real number line.
– Both values are strings; x comes before y in dictionary order (assuming a case-sensitive order using the ASCII encoding of characters).
– Both values are user constants; the name of x comes before the name of y in dictionary order.
– Both values are complex, i.e. $x = f(t_1, \ldots, t_n)$ and $y = g(s_1, \ldots, s_m)$, and any of the following are satisfied:
  • The name of constructor f comes before the name of the constructor g in dictionary order.
  • Both constructors have the same name, and the first i where $t_i \neq s_i$ then $t_i < s_i$.

Here are some examples:

```
0 < 1, 1 < "FALSE", "FALSE" < FALSE, FALSE < TRUE
```

Table 3. Table of built-in data types

Built-in Data Types	
Name	Meaning
`Real`	The set of all numeric values.
`Integer`	The set of all integers.
`Natural`	The set of all non-negative integers.
`PosInteger`	The set of all positive integers.
`NegInteger`	The set of all negative integers.
`String`	The set of all string integers.
`Boolean`	The set of constants `TRUE` and `FALSE`.

```
Node(1, Node(10, NIL, NIL), NIL) < Node(2, NIL, NIL),
    Node(2, NIL, NIL) < Person("John", "Smith")
```

The predefined relations <, <=, >, and >= use this order. The predefined functions `min` and `max` find the smallest and largest values according to <. Finally, all predefined functions that must sort values, e.g. `toList`, also use this order.

1.4 Data Types and Subtyping

A *data type* (or just a *type*) is a expression standing for a set of values. Table 3 lists the built-in data types and their meanings. In addition, other types can be defined. Suppose $f(...)$ is an n-ary constructor, then the type f stands for the range of the constructor f. Suppose c is a constant, then the type $\{c\}$ stands for the singleton set containing c. Suppose τ_1 and τ_2 are types, then $\tau_1 + \tau_2$ stands for the set-union of the two types. Finally, $f(\tau_1, \ldots, \tau_n)$ stands for the set of all values obtained by applying f to all possible values in $\tau_1, \ldots, \tau_n$. FORMULA provides some special syntax to make it easier to write types. A *finite enumeration* is a set of constants:

```
{ RED, GREEN, FALSE, 1, 2, "Hello" }
```

An enumeration can also include integer ranges:

```
{ -1000..1000, 1001..1001, 1002 }
```

This type stands for the set of all strings, integers, and Boolean values:

```
Real + String + { TRUE, FALSE }
```

This type stands for the set of all integer-keyed binary trees with a non-empty left child:

```
Node(Integer, Node, Node + { NIL })
```

Also, this type stands for the set of all integer-keyed binary trees with a non-empty right child:

```
Node(Integer, Node + { NIL }, Node)
```

Data types are related to each other by the *subtyping relation*. In object-oriented languages subtyping is indicated by explicitly subclassing a base class, extending an interface, or implementing an interface. In FORMULA the subtyping relationship is determined implicitly by the values a type represents. A type τ_1 is a subtype of τ_2 if the values represented by τ_1 are a *subset* of those represented by τ_2. We write τ_1 `<:` τ_2 if τ_1 is a subtype of τ_2. Here are some examples of types satisfying the subtyping relationship:

```
{ 1, 2 } <: PosInteger <: Natural + String <: Real + String
```

```
Node(Integer, Node, Node)<: Node(Integer, Node, Node + {NIL})
```

```
Node(Integer, Node, Node + {NIL}) <: Node
```

The above examples show that types are fairly precise; they can represent very specific sets of data. Also, types are *behavioral*; FORMULA only cares about the set of values a type stands for, but not how the type is written. For example, all of these types mean the same:

```
{ 0..10 } + Natural = Natural = {0} + PosInteger
```

```
Node(Integer, Node + {NIL}, Node + {NIL}) = Node
```

FORMULA infers types for expressions, and these types over-approximate the evaluation of expressions. We write $e : \tau$ if the expression e is assigned the type τ. For instance, a *C++* compiler might assign the *int* type to the expression 1 or the *float* type to the expression 1.5. Because FORMULA types are more precise, the type of a value is just a singleton set containing that value, i.e. $1 : \{1\}$ and $1.5 : \{\frac{3}{2}\}$. Because subtyping is implicit by subset inclusion, the value 1 can be used anywhere a *Real* is accepted without coercion. This is unlike *C++*, where the integer value 1 would be coerced to the floating point value 1.0 because *int* and *float* are different kinds of values.

Consider the more complicated *C++* example:

```
enum E { Zero = 0, One = 1, Two = 2 };
bool Foo(E x, E y)
{
   auto z = x + y;
   return z > 10;
}
```

The *C++* compiler infers $z : int$, even though the expected values for x and y are in the interval $[0, 2]$. Also, z is always less than 10 and $z > 10$ is always false. Consider an analogous FORMULA specification:

```
1: transform Foo(x: E, y: E) returns (b:: Bool)
2: {
3:    E ::= { 0..2 }.
4:    Return(TRUE) :- z = %x + %y, z > 10.
5: }
```

The *transform* takes two parameters x and y of type E (defined in line 3). The rule in line 4 is triggered whenever $z = x + y$ and $z > 10$. FORMULA infers that $x, y : \{0..2\}$, $z : \{0..4\}$, $z > 10 : \{\texttt{FALSE}\}$. Finally, it issues an error because the condition $z > 10$ can never be satisfied. (We explain the structure of rules in the next section.) Thus, type inference can be used to catch errors in specifications.

1.5 Type Declarations

Type declarations are used to: (1) define simple names for complicated type expressions, (2) introduce new data constructors along with the types of their arguments, (3) introduce new user-defined constants. Type declarations come in two forms. The first form assigns a name to a type expression.

```
TypeName ::= TypeExpr.
```

The second form introduces a new data constructor.

```
ConstructorName ::= (Arg1_TypeExpr, ..., ArgN_TypeExpr).
```

The expressions appearing in declarations are restricted; they cannot contain constructor applications such as $Node(Integer, NIL, NIL)$. However, it is legal to use the type $Node$, which stands for the entire range of the $Node(,,)$ constructor. This restriction allows for more efficient type inference and type manipulation.

Type declarations must be placed in modules, which are self-contained units. The meaning of a type declaration is understood w.r.t. all the type declarations within the same module. In the examples to follow we use *domain modules* to hold type declarations. For now it is enough to understand that type declarations are not visible outside of their modules. (We describe domains in detail in the next section.) Below are two modules D and D' that define the type Id in different ways:

```
domain D { Id ::= Integer. } domain D' { Id ::= String. }
```

In domain D the type Id stands for integers and in D' it stands for strings. The FORMULA compiler accepts these declarations because they occur in two distinct modules. On the other hand these declarations are illegal.

Error: Conflicting definitions of type Id

```
domain D
{
  Id ::= Integer.
  Id ::= String.
}
```

There are multiple conflicting declarations of the type *Id* in the same module. FORMULA accepts any set of type declarations as long as their *meaning* is consistent across the module. For example, this module is legal because it defines *Id* in two equivalent ways.

Legal: All definitions of type Id are equivalent

```
domain D
{
  Id ::= Integer.
  Id ::= NegInteger + {0} + PosInteger.
}
```

1.6 Declaring Constants

User-defined constants are implicitly declared by using them in some enumeration in some type declaration. Every domain automatically contains the declaration:

```
Boolean ::= { TRUE, FALSE }.
```

Introduces user-defined constants RED, GREEN, and BLUE.

```
domain Colors
{
  NamedColor ::= { RED, GREEN, BLUE }.
  Color      ::= { RED, GREEN, BLUE, 0..16777215 }.
}
```

The *NamedColor* type contains the constants *RED*, *GREEN* and *BLUE*. These constants are implicitly declared by using them in the type declaration. The *Color* type also mentions these constants along with all integers in the 24-bit RGB color spectrum. User-defined constants are only distinguished by name, so every occurrence of *RED* stands for the same user-defined constant. Unlike *C++*, a user-defined constant is not equivalent to an integer.

In C++ user-defined constants are actually integers; not possible in FORMULA.

```
 enum Color { RED = 0xFF0000, GREEN = 0x00FF00, BLUE = 0x0000FF };
```

Unlike *C#*, user-defined constants exist independently of the type declaration in which they are introduced.

```
 C# introduces constants NamedColors.RED, NamedColors.GREEN,
NamedColors.BLUE
  enum NamedColor { RED, GREEN, BLUE }
 C# introduces constants Colors.RED, Colors.BLUE, Colors.GREEN
  enum Color { RED, GREEN, BLUE }
```

1.7 Declaring Data Constructors

Data constructors are declared by special syntax. The left-hand side of the declaration is the name of the constructor and the right-hand side is a comma-separated list of argument types with parenthesis. Every constructor must have at least one argument, otherwise it would be constant. Here is another domain providing constructors for colors:

```
1: domain Colors
2: {
3:   NamedColor ::= (String).
4:   RGBColor   ::= (r: {0..255}, g: {0..255}, b: {0..255}).
5:   RGBAColor  ::= (a: {0..255}, r: {0..255}, g: {0..255}, b: {0..255}).
6:   Color      ::= NamedColor + RGBColor + RGBAColor.
7: }
```

NamedColor (line 3) defines a unary constructor taking a string. This constructor can be used to create values such as:

```
NamedColor("RED"), NamedColor("GREEN"), NamedColor("BLUE")
```

It is illegal to apply the *NamedColor* color constructor to values other than strings. Also the corresponding *NamedColor* type is automatically defined and only contains those values that obey argument types. The *RGBColor* constructor (line 4) takes three arguments for the red, green, and blue components. The arguments have been given explicit names r, g, and b. Naming arguments is optional but useful. Finally, the *Color* type is a union of the possible color values. As before, there are no implicit conversions between values. For instance:

```
RGBColor(0, 0, 255) != 255 != RGBAColor(0, 0, 0, 255).
```

Every constructor creates distinct values.

Data constructors are similar to *structs* or records in *C*-like languages, but more general. Consider the task of defining a node struct in *C#*. The following code is illegal because the struct *Node* directly depends on *Node* values.

```
Cannot define a struct that depends directly on itself.
struct Node
{
   int key; Node left; Node right;
   Node(int k, Node l, Node r)
   { key = k; left = l; right = r; }
};
```

The problem is that *Node* must have a default value, and there is no way to construct this default value. The definition becomes legal if *struct* is replaced with *class* and then *null* is a valid default value for the *left* and *right* fields. However, this comes with the price that node equality is significantly weakened, i.e. $n == m$ only if the variables n and m hold the same reference. In other words, binary trees cannot be compared as if they were just values.

The declarations of FORMULA constructors can cyclically depend on themselves. Here is the equivalent definition for a *Node* in FORMULA.

```
domain Trees
{
   Node ::= (key  : Integer,
             left : Node + {NIL},
             right: Node + {NIL}).
}
```

The only requirement on constructors is that there must be some arguments of finite size satisfying the type constraints of the constructor. For example, a minimal node value can be constructed by:

```
Node(0, NIL, NIL)
```

Note that *NIL* is not a keyword; just a user-defined constant. This specification has an error because there is no way to construct a node value using a finite number of applications.

```
domain Trees
{
   Node ::= (key  : Integer,
             left : Node,
             right: Node + {NIL}).
}
```

The problem is the *left* field can only take a node value, but the only way to construct a node value is to apply the node constructor. Therefore, only an infinitely long sequence of node applications could construct such a value. FORMULA returns an error message like this:

```
(3, 4): The type Node is badly defined; it does not accept
        any finite terms.
```

The following domain defines nodes in two equivalent ways. It is accepted by the compiler:

```
domain Trees
{
   Node ::= (key  : Integer,
             left : Node + {NIL},
             right: Node + {NIL}).

   Node ::= (key: Integer, left: Tree, right: Tree).
   Tree ::= Node + {NIL}.
}
```

The type *Tree* is a super-type of the type *Node*, because in contains all node values and the additional value *NIL*.

2 Domains and Models

The purpose of a domain is describe a "class of things". The purpose of a model is to describe a specific "thing". Here are a few examples that we explore in this tutorial:

1. **DAG**: The DAG domain describes the properties of directs acyclic graphs (DAGs). A DAG model represents an individual DAG.
2. **SAT**: The SAT domain describes the set of satisfiable boolean expressions. A SAT model describes a single expression and the variable assignments that witness its satisfiability.
3. **FUNC**: The FUNC domain describes a small language of arithmetic functions and the rules of their evaluation. A FUNC model represents a program of the FUNC language.

The first step to define a "class of things" is to create a representation for "things" using FORMULA data types. Consider the classical definition of a directed graph G:

$$G \overset{def}{=} (V, E) \qquad \text{where } E \subseteq V \times V. \tag{1}$$

Classically, a directed graph is represented by a set of vertices V and set of edges E; each $e \in E$ is a pair of vertices. Furthermore, suppose vertices are represented by integers, then the set of all finite integer-labeled graphs is:

$$\mathcal{G} \overset{def}{=} \{(V, E) \mid V \subset \mathbb{Z} \wedge E \subseteq V \times V\} \qquad \text{where every } V \text{ is finite.}$$

Here is an example of a specific graph:

$$G_{ex} \overset{def}{=} (\{1, 2, 100\}, \{(1, 2), (100, 100)\}).$$

FORMULA does not directly support sets and relations, so we cannot express vertices and edges as sets. Instead, integer-labeled graphs are represented using two data constructors V and E as follows:

Example 1 (Integer-labeled graphs).

```
domain IntGraphs
{
   V ::= new (lbl: Integer).
   E ::= new (src: V, dst: V).
}
```

Intuitively, vertices are values such as:

```
V(1), V(2), V(100)
```

and edges are values such as:

```
E(V(1), V(2)), E(V(100), V(100))
```

Though these constructors provide representations for the elements of a graph, the domain does not define any specific graph elements. This is because the domain is intended as a schema for all graphs; it is not intended to represent a specific graph. A specific graph is represented by a model, as follows:

Example 2 (A small graph).

```
1: model Gex of IntGraphs
2: {
3:    V(1).
4:    V(2).
5:    V(100).
6:    E(V(1), V(2)).
7:    E(V(100), V(100)).
8: }
```

A model has a name and indicates the domain to which it belongs (line 1). The body of a model is a list of values separated by periods. These periods are actually assertions about the model. Each expression $f(\ldots)$. is an assertion:

"The value $f(\ldots)$ is always provable in the model M."

Thus, the model *Gex* contains a set of assertions, built with data constructors, about some vertex and edge values. These assertions define the specific elements of the graph *Gex*.

2.1 Querying Models

To understand how a model contains a set of assertions, create a file called `ex1.4ml` and copy the code contained in Examples 1 and 2. A *query operation* tests if a property is provable on a model. Follow these steps to test if a vertex called $V(1)$ exists in the model *Gex*:

```
1:  ...\Somewhere>Formula.exe
2:
3:  []> load ex1.4ml
4:  (Compiled) ex1.4ml
5:  0.82s.
6:  []> query Gex V(1)
7:  Started query task with Id 0.
8:  0.06s.
9:  []> ls tasks
10:
11: All tasks
12:  Id | Kind  | Status | Result |      Started       | Duration
13: ----|-------|--------|--------|--------------------|----------
14:  0  | Query |  Done  |  true  | 5/17/2013 3:28 PM |  0.04s
15: 0.03s.
```

Line 3 loads and compiles the file `ex1.4ml`. Line 6 starts a query operation on the model *Gex* and tests for the property $V(1)$. If $V(1)$ is provable in *Gex* then this query operation returns true; otherwise it returns false. Starting a query spawns a new background task that may take some time to complete. The Id of the newly created task is reported (e.g. Id 0). Line 9 causes the status of all tasks to be displayed. Line 14 shows that the query completed with the result true.

On the other hand this query evaluates to false, because there is no vertex named $V(3)$:

```
[]> query Gex V(3)
```

This is very important: Any query that is not provable using the model (and its domain declarations) is false. The query $V(3)$ is false for the model *Gex* because there are no assertions that can prove it. We can ask more interesting queries, such as: Does there exist some vertex?

```
[]> query Gex V(x)
```

This query contains a *variable* called x. More generally, a query is true if there is some substitution for the variables that is provable. This query is true because replacing x with 1, 2, or 100 forms queries that are provable. Variables appearing in a query are not declared and are local to the query expression.

More complicated patterns can be used in a query. This one tests if there is an edge that loops back onto the same vertex.

```
[]> query Gex E(x, x)
```

It is true because if $x = V(100)$ then $E(V(100), V(100))$ is provable. As with any language, it is possible to mistype commands. The FORMULA type system will catch some of these mistakes. For example, the query

```
[]> query Gex E(V(x), x)
```

is always false because x must simultaneously be an integer and a vertex, which is never possible. In this case the query is ignored and warning messages are returned:

```
[]> qr Gex E(V(x), x)
commandline.4ml (2, 1): Argument 2 of function E is badly typed.
commandline.4ml (0, 0): The install operation failed
Failed to start query task.
0.02s.
[]>
```

Queries can be conjoined using the comma operator (','). Such a query is true if there is some substitution of variables making every conjunct true. This query tests if there are two edges that can be placed end-to-end:

```
[]> query Gex E(x, y), E(y, z)
```

Notice that the variables x and z only appear once in the query. Variables that only appear once can be written with an underscore ('_'); every occurrence of an underscore creates a new variable with a different name from all other variables in the expression. The previous query can be rewritten as:

```
[]> query Gex E(_, y), E(y, _)
```

Underscores are useful for visually emphasizing those variables that appear in multiple places. Queries can also be formed from built-in relations and constraints.

```
[]> query Gex V(x), x > 20
```

Constructor labels can be used to write more readable queries. The query:

"Is there an edge whose source vertex has a label greater than 100?"

can be written as:

```
[]> query Gex e is E, e.src.lbl > 100
```

The constraint e *is* E requires e to be a provable value of type E.

The order in which conjuncts are written does not matter; the semantics of a query is the same. However, queries can only check for properties that can be answered using model assertions and a finite number of evaluations. For example, this query is not allowed:

```
[]> query Gex x > 100
```

It asks if there exists a number greater than 100. While the answer is "yes", it cannot be proved using the assertions written in the model, nor can it be proved by evaluating > for a finite number of values. In this case, the following message is returned:

```
[]> query Gex x > 100
commandline.4ml (2, 3): Variable x cannot be oriented.
commandline.4ml (0, 0): The install operation failed
Failed to start query task.
0.02s.
[]>
```

The message `‘‘Variable x cannot be oriented’’` means the compiler cannot express x in such a way that the query can be evaluated. Note that FORMULA can reason about the properties of numbers, but the query operation is not the mechanism to accomplish this. We shall discuss this more in Section ??.

Finally, it is important to remember that even though $f(\dots, t, \dots)$ might be provable, this does not imply that t is itself provable. Consider the following model:

```
model NoVertices of IntGraphs
{
   E(V(1), V(2)).
}
```

This query is true:

```
[]> query NoVertices E(_,_)
```

But this query is false:

```
[]> query NoVertices V(_)
```

There are no provable vertices, even though a vertex value does appear within a provable edge. (This may surprise users familiar with term-rewriting systems.)

2.2 Model Conformance

A model M *conforms* to a domain D if the following properties are satisfied:

P1. Every assertion in M is constructed from *new-kind* constructors.
P2. Every constructor application in M obeys the type declarations in D.
P3. The operation `query M D.conforms` evaluates to true.

P1-P2 are checked whenever a model is compiled; any violation causes a compile-time error; P3 is checked upon request. A *new-kind* constructor is a constructor marked with the modifier *new*. Recall that both V and E constructors were marked with this modifier.

```
V ::= new (lbl: Integer).
E ::= new (src: V, dst: V).
```

Constructors that are not marked with the *new* modifier are only used to perform auxiliary computations; they can never appear in models. (We demonstrate this in more detail in later sections.) P2 is familiar from earlier examples. It is always illegal to use constructors with badly typed arguments, as in:

```
model BadlyTyped of IntGraphs
{
   E("Foo", "Bar").
}
```

P3 allows domains to place fine-grained constraints on the conformance relationship. Unlike the first two properties, satisfying P3 can be difficult and evaluating its satisfaction can be expensive. For these reasons, FORMULA only checks P3 upon request and failure of this property is not an error.

2.3 Relational Constraints

We began by trying to express the set of all integer-labeled finite graphs. The intent was to define *IntGraphs* so its set of conforming models would be equivalent to the set of integer-labeled finite graphs. Properties P1-P2 guarantee that models contain only vertex and edge assertions, which encode graph elements in an obvious way. However, there is the additional constraint that graph edges should be pairs of graph vertices, i.e. $E \subseteq V \times V$. Did we capture this constraint correctly? The answer is that it depends on how we choose to define the set of vertices V present in a model M. There are two options: (1) Every occurrence of a vertex value anywhere in the model is implicitly a member of V. (2) Only those vertices that are provable are members of V. Consider this example:

```
model SomeVertices of IntGraphs
{
   V(1).
   E(V(1), V(2)).
}
```

Under the first definition, the vertex set for this model is $\{1, 2\}$ and the edge set is $\{(1, 2)\}$. This satisfies the constraint $E \subseteq V \times V$. Under the second definition the vertex set is only $\{1\}$ and the edge set violates the constraint because $2 \notin V$. By default, FORMULA uses the second and more restrictive definition: $E \subseteq V \times V$ means every argument to $E(,)$ must be provable. Models violating this property do not conform to the domain. These kinds of constraints are so common that FORMULA automatically introduces them. To see this, add the code for *SomeVertices* to `ex1.4ml` and run this query:

```
[]> query SomeVertices IntGraphs.conforms
```

The result of this query is false because $V(2)$ is not derivable. Add $V(2)$ to the model, save it, and type:

```
[]> reload ex1.4ml
```

Evaluate the query again to observe that it evaluates to true.

We call the previous kind of constraint a *relational constraint*. Relational constraints are injected by the compiler when it appears that one constructor is being used to encode a set S and another constructor is being used to encode a relation $R \subseteq \ldots \times S \times \ldots$.

Definition 21 (Relational Constraint)**.** The constructor R is relational on constructor S in position i if the constructor R is declared as:

```
R ::= new (..., arg_i: T_i, ...).
```

and S is a subtype of T_i. The relational constraint means that for every provable value t containing $R(\ldots, t_i, \ldots)$ and $t_i = S(\ldots)$ then t_i must also be provable.

Some type declarations are not intended to encode finite relations, and the default behavior would produce strange results. Consider the following recursive definition for binary trees:

```
Node ::= new (left: Node + {NIL}, right: Node + {NIL}).
```

This declaration fits the pattern for relational constraints and the generated constraints are satisfiable. However, it assumes the user intended to encode a relation with the following strange property:

$$Node \subseteq Node \times Node.$$

The only finite relation satisfying this property is the empty set. There is clearly a semantic mismatch between the binary relation $Node$ and the binary data constructor $Node$. In order to bring attention to this mismatch FORMULA produces the following error:

```
ex.4ml (3, 4): The constructor Node cannot have relational
               constraints on itself; see argument 1.
ex.4ml (3, 4): The constructor Node cannot have relational
               constraints on itself; see argument 2.
```

When this error occurs at position i, the user must explicitly indicate that the constructor is not intended to encode a relation at position i. Placing the *any* modifier before the argument type indicates that *any* well-typed value is permitted here, not just those values that are provable. The compiler does not generate relational constraints for this argument and the error message does not occur. Here is the proper declaration for the recursive binary tree constructor:

```
Node ::= new (left: any Node + {NIL}, right: any Node + {NIL}).
```

Of course, binary trees can also be encoded using finite relations, but the encoding is a different one from the recursive constructor shown above.

2.4 Finite Functions

Finite functions are a special case of finite relations satisfying additional constraints. Consider the representation of a forest F of binary trees as follows:

$$F \overset{def}{=} (V, parent) \qquad \text{where } parent : V \to \{\top\} \cup (\{L, R\} \times V).$$

The set V contains the vertices of the forest and the function *parent* assigns to each vertex v its parent: $parent(v) = (L, u)$ if v is the left child of u; $parent(v) = (R, u)$ if v is the right child of u; $parent(v) = \top$ if v is a root. Additionally, the *parent* function should not introduce a cycle, but we ignore this constraint for now. The parent function can be treated as a finite relation by listing its input-output pairs. Consider this forest where 1 is a root, 2 is its left child, and 3 is its right child:

$$F_{ex} \overset{def}{=} (\{1, 2, 3\}, \{(1, \top), (2, (L, 1)), (3, (R, 1))\}).$$

The *parent* relation encodes a total finite function if it is total on the domain:

$$\forall v \in V.\ \exists x.\ (v, x) \in parent,$$

and every input is related to a unique output:

$$\forall v \in V.\ \forall x, y.\ (v, x) \in parent \land (v, y) \in parent \Rightarrow x = y.$$

FORMULA supports declarations of finite functions also. As before, these declarations are really introducing data constructors plus additional constraints that provable values must encode finite functions. Example 3 shows the syntax.

Example 3 (Relational Trees).

```
domain RelTrees
{
   V       ::= new (lbl: Integer).
   Parent  ::= fun (chld: V => cxt: any {ROOT} + Context).
   Context ::= new (childPos: {LFT, RT}, prnt: V).
}

model Fex of RelTrees
{
   V(1). V(2). V(3).
   Parent(V(1), ROOT).
   Parent(V(2), Context(LFT, V(1))).
   Parent(V(3), Context(RT, V(1))).
}
```

Line 5 declares the *Context* constructor to represent $\{L, R\} \times V$. Line 4 declares the *Parent* constructor using the *fun* modifier. This modifier implies *new*. The arguments on the left side of => correspond to the domain of the relation and the arguments on the right side correspond to the codomain. The codomain contains the value *ROOT* (i.e. $\top$) and any *Context* value. The *fun* modifier injects uniqueness constraints, i.e. for all provable values $Parent(v, x)$ and $Parent(v, y)$ then $x = y$. The *totality arrow* => injects totality constraints, i.e. for every provable $V(x)$ there is a provable $Parent(V(x), y)$. Totality is affected by the *any* modifier. If an argument is marked with *any*, then the finite function must be defined for every well-typed value. For example, this declaration would cause an error:

```
Parent  ::= fun (chld: any V => cxt: any {ROOT} + Context).
```

The *any* modifier applied to the first argument means there must be a provable $Parent(V(x), y)$ value for every well-typed value $V(x)$. This implies an infinite

Table 4. Table of relation / function modifiers

Relation / Function Modifiers	
Syntax	Meaning
`R ::= new (..., T_i, ...).`	Relational constraint: If S is a constructor, $S <: T_1$, $R(\ldots, t_i, \ldots)$ occurs in a provable value, and $t_i = S(\ldots)$, then t_i must be provable.
`R ::= new (..., any T_i, ...).`	Occurrences of R are exempt from the relational constraint in position i.
`R ::= fun (..., D_m -> ..., C_n).`	Partial function: Same as `new`. Additionally, the set of provable R values must form a partial function from $D_1 \times \ldots \times D_m$ to $C_1 \times \ldots \times C_n$.
`R ::= fun (..., D_m => ..., C_n).`	Total function: Same as partial function, but must be total on $D_1 \times \ldots \times D_m$; totality is modified by `any`.
`R ::= inj (..., D_m -> ..., C_n).`	Partial injection: Same as partial function; additionally if $R(\boldsymbol{x}, \boldsymbol{z})$ and $R(\boldsymbol{y}, \boldsymbol{z})$ are provable then $\boldsymbol{x} = \boldsymbol{y}$.
`R ::= inj (..., D_m => ..., C_n).`	Total injection: Constrained to be a total function and partial injection.
`R ::= sur (..., D_m -> ..., C_n).`	Partial surjection: Same as partial function; additionally must be total on $C_1 \times \ldots \times C_n$ (totality is modified by `any`).
`R ::= bij (..., D_m -> ..., C_n).`	Bijection: Constrained to be a total surjection and partial injection; partiality arrow has no effect.
`R ::= bij (..., D_m => ..., C_n).`	Bijection: Constrained to be a total surjection and partial injection.

number of provable values, which is not permitted. FORMULA returns the following error message:

```
ex.4ml (4, 4): The function Parent requires totality on an
               argument supported by an infinite number of
               values; see argument 1.
```

Whenever the domain of a finite function is infinite that function cannot be total, but it can be *partial*. The *partial arrow* −> indicates a partial function, which need not be defined on every element of its domain. Suppose every vertex label should also be given a "pretty name". This could be expressed by extending the signature of V as follows:

```
V ::= fun (lbl: Integer -> prettyName: String).
```

Every vertex must have a unique pretty name, but there does not need to be a vertex defined for every integer. Table 4 shows the complete variety of relation / function modifiers.

2.5 Recursive Types, Aliases, and Symbolic Constants

Example 3 showed a partial specification of binary trees using a representation inspired by finite relations. Another approach is to use the full power of recursive data types. In this representation an entire tree is a single complex value. The locations of values distinguishes vertices from each other. This is in contrast to using of unique identifiers to distinguish vertices.

Example 4 (Algebraic Trees).

```
1:  domain AlgTrees
2:  {
3:     Node ::= new (left:  any Node + {NIL},
4:                   right: any Node + {NIL}).
5:     Root ::= new (root:  any Node).
6:  }
7:
8:  model Fex' of AlgTrees
9:  {
10:    Root(
11:       Node(
12:            Node(NIL, NIL),
13:            Node(NIL, NIL))).
14: }
```

Example 4 shows an algebraic representation of trees using data constructors. Notice that the entire tree is a single value (line 10). The *Root*() constructor is used to mark some nodes as roots in the forest. The left and right children of the root are both nodes without children, represented by the value $Node(NIL, NIL)$ (lines 12, 13). In fact, both these nodes are exactly the same value. They are distinguishable by where they occur in the construction of the parent. This representation has several advantages: (1) Nodes do not need to be labeled. (2) It is impossible to create an illegal tree. (3) Two trees are the same if and only if they are the same value. Also, because the same value can represent many nodes, it is possible to define the value once and reuse it in many places. Reuse is accomplished by introducing an *alias* as follows:

```
leaf is Node(NIL, NIL).
```

The right-hand side of the *is* keyword requires a constructed assertion. The left-hand side is an identifier that stands for the constructed value. Using aliases, the previous model can be expressed as:

```
model Fex_Shared of AlgTrees
{
   leaf is Node(NIL, NIL).
   Root(Node(leaf, leaf)).
}
```

Aliases can be used to represent models with exponentially less space. Consider the following complete binary tree with 1,023 nodes:

```
model BiggerTree of AlgTrees
{
   leaf is Node(NIL, NIL).

   subtree_3    is Node(leaf       , leaf).
   subtree_7    is Node(subtree_3  , subtree_3).
   subtree_15   is Node(subtree_7  , subtree_7).
   subtree_31   is Node(subtree_15 , subtree_15).
   subtree_63   is Node(subtree_31 , subtree_31).
   subtree_127  is Node(subtree_63 , subtree_63).
   subtree_255  is Node(subtree_127, subtree_127).
   subtree_511  is Node(subtree_255, subtree_255).
   subtree_1023 is Node(subtree_511, subtree_511).

   Root(subtree_1023).
}
```

Aliases are visible to all assertions within their defining model. The order in which aliases are defined is inconsequential. However, aliases definitions cannot form as cycle as this would be equivalent to applying an infinite number of constructors. This property is checked at compile time. Here is an example:

```
model InfiniteTree of AlgTrees
{
   infinite_left  is Node(infinite_right, NIL).
   infinite_right is Node(NIL, infinite_right).
   Root(Node(infinite_left, infinite_right)).
}
```

The *infinite_left* node has an infinitely deep subtree as its left node, and the *infinite_right* node has an infinitely deep subtree as its right node. This problem is detected and the following error is reported.

```
inf.4ml (4, 4): Symbolic constant InfiniteTree.%infinite_right is
                defined using itself.
inf.4ml (3, 4): Symbolic constant InfiniteTree.%infinite_left is
                defined using itself.
```

2.6 Symbolic Constants

The previous error messages referred to *symbolic constants*. A symbolic constant is a *constant*, i.e. it is a function that takes no arguments and returns a value. However, the value it returns may not be known at compile time. This latter property is unlike the constants we have encountered so far. For example, the constant 1 always returns the constant 1 and the constant *NIL* always returns the constant *NIL*. Each model alias a declared in model M defines a symbolic constant called $M.\%a$. It returns the value produced by expanding all aliases in the model. Consider Example 2 rewritten with aliases:

```
model Gex' of IntGraphs
{
   v1   is V(1).
   v2   is V(2).
   v100 is V(100).

   e_1_2     is E(v1, v2).
   e_100_100 is E(v100, v100).
}
```

These aliases create symbolic constants with the following properties:

$$\begin{aligned}
&Gex'.\%v1 &&= V(1)\\
&Gex'.\%v2 &&= V(2)\\
&Gex'.\%v100 &&= V(100)\\
&Gex'.\%e\_1\_2 &&= E(V(1), V(2))\\
&Gex'.\%e\_100\_100 &&= E(V(100), V(100))
\end{aligned}$$

Unlike variables, symbolic constants can be used to test the properties of specific model elements. Whenever a symbolic constant occurs in a query, it matches the corresponding assertion in the model where it was defined. For example:

```
[]> query Gex' Gex'.%v1.lbl = 1
```

This query is satisfied if the value represented by *Gex'*.%*v*1 is provable and its label is equal to 1. This query evaluates to true. Symbolic constants are prefixed with the percent-sign ('%') to distinguish them from variables. Consider this query:

```
[]> query Gex' E(%v1, v1)
```

Here %*v*1 is the same symbolic constant as before, but *v*1 is a variable. (Names do not need to be fully qualified if they can be unambiguously resolved.) This query is satisfied if there is some value for *v*1 that makes $E(V(1), v1)$ provable. The assignment $v1 = V(2)$ is one such value and the query evaluates to true. Finally, this query evaluates to false, because there is not an edge from $V(1)$ to $V(1)$:

```
[]> query Gex' E(%v1, %v1)
```

2.7 Separate Compilation

A FORMULA module can refer to modules in other files. The compiler will load and compile all files required to completely compile a program. There are several ways to refer to modules in another files. The first way is to qualify the module name with a source file using the *at* operator. For example:

```
model M of D at "foo.4ml" { ... }
```

The compiler will look for a domain named *D* in the file *foo.4ml*. This method does not affect the resolution of any other occurrence of *D*. For example, this code loads two different domains, both called *D*, from different files:

```
model M1 of D at "..\\..\\version1.4ml" { ... }
model M2 of D at '"..\..\version2.4ml"' { ... }
```

(Relative paths are resolved w.r.t. to the path of file where they occur.) However, it is illegal to define two modules with the same name in the same file.

```
model M of D at "version1.4ml" { ... }
model M of D at "version2.4ml" { ... }
```

This causes an error:

```
ex.4ml (2, 1): The module M has multiple definitions.
               See ex.4ml (1, 1) and ex.4ml (2, 1)
```

The disadvantage of the *at* operator is that it must be used on every reference to *D*. Another option is to register *D* using a *configuration block*. Configuration blocks have access to various configuration objects, one of which is called *modules* mapping names to files. To register *D* and *D'* in the scope of the file write:

```
1: [
2:    modules.D  = "D  at foo.4ml",
3:    modules.D' = "D' at foo.4ml"
4: ]
5:
6: model M of D { ... }
```

Lines 1 - 4 form a configuration block. Now every occurrence of D will resolve to D located in *foo.4ml*. If D is defined to be in several locations, then an error occurs.

Causes an error because D is defined to be at two different places.

```
[
   modules.D = "D at version1.4ml",
   modules.D = "D at version2.4ml"
]
```

Finally, each module has its own local configuration parameters. Modules inherit file level configurations, and can extend these with additional parameters. Module-level configurations are isolated from each other, so occasionally it may be useful to register domains at this level. Here is an example:

```
model M1 of D
[
   modules.D = "D at version1.4ml"
]
{
   ...
}

model M2 of D
[
   modules.D = "D at version2.4ml"
]
{
   ...
}
```

The module-level configuration block must be placed after the module declaration and before the opening curly brace. These configurations do not conflict, because they are lexically scoped to the modules $M1$ and $M2$. Separate compilation is available for all types of FORMULA modules.

3 Rules and Domain Constraints

Domain constraints are essential for defining "classes of things". So far we have demonstrated type constraints and a few kinds of relation / function constraints (that also appear in type declarations). However, these constraints are not general enough to capture many properties. Recall the relational trees example

(Example 3) where we neglected to constrain trees to be acyclic. There was no way to write this constraint using the mechanisms presented thus far, and so the specification is incomplete. Some models that are not trees conform to the *RelTrees* domain.

The remaining aspects of the FORMULA language deal with computing properties of models using conditional statements, which we call *rules*. These computed properties can be used to write powerful domain constraints (among other things). A basic rule has the following syntax:

```
head :- body.
```

The *body* part can contain anything a query can contain. The *head* part is a sequence of constructor applications applied to constants and variables. A rule means:

"Whenever the body is provable for some variable assignment, then the head is also provable for that variable assignment."

Thus, a rule is a logical statement, but it can also be *executed* (like a query) to grow the set of provable values.

Consider the problem of computing the ancestors between vertices in a *RelTrees* model. First, we introduce a helper constructor called *anc*(,) into the *RelTrees* domain to represent the ancestor relationship:

```
anc ::= (ancestor: V, descendant: V).
```

Notice that *anc* is not modified with *new* (or any other modifier that implies *new*). This means an *anc* value can never be asserted by a model. The only legal way to prove an *anc* value is by proving its existence with rules. We call *anc* a *derived-kind* constructor. Here is a rule that can prove u is an ancestor of w if u is the parent of w:

```
anc(u, w) :- Parent(w, Context(_, u)).
```

Here is a rule that can prove u is an ancestor of w if u is an ancestor v and v is an ancestor of w.

```
anc(u, w) :- anc(u, v), anc(v, w).
```

As with queries, the comma operator (',') conjuncts constraints. Every conjunct must be satisfied for the rule to be satisfied. If two rules have the same head, then they can be syntactically combined using the semicolon operator (';').

```
anc(u, w) :- Parent(w, Context(_, u)); anc(u, v), anc(v, w).
```

Intuitively the semicolon operator behaves like disjunction, but remember that each semicolon actually marks the body of an independent rule.

Table 5. Table of matching constraints

Matching Constraints	
Syntax	Meaning
`q`	true if the derived constant q is provable; false otherwise.
`f(t1,...,tn)`	true for every assignment of variables where $f(t_1, \ldots, t_n)$ is provable; false otherwise.
`x is f(t1,...,tn)`	true for every assigment of variables where $f(t_1, \ldots, t_n)$ is provable and $x = f(t_1, \ldots, t_n)$; false otherwise.
`x is T`	true if x is a provable value and a member of the type named T; false otherwise.

Table 6. Table of interpreted predicates

Interpreted Predicates	
Syntax	Meaning
`no` $\{\ldots\}$	true if the set comprehension is empty; false otherwise.
`t = t'`	true if t and t' are the same value; false otherwise.
`t != t'`	true if t and t' are different values; false otherwise.
`t < t'`	true if t is less than t' in the order of values; false otherwise.
`t <= t'`	true if t is less than equal to t' in the order of values; false otherwise.
`t > t'`	true if t is greater than t' in the order of values; false otherwise.
`t >= t'`	true if t is greater than or equal to t' in the order of values; false otherwise.
`x : T`	true if x is a member of the type named T; false otherwise.

3.1 Derived Constants

A tree-like graph is not a tree if it has a cycle in its ancestor relation. There is a cycle if and only if some vertex is an ancestor of itself. The following rule summarizes this property:

```
hasCycle :- anc(u, u).
```

The symbol *hasCycle* is a *derived-kind* constant (or just *derived constant*). A derived constant is declared by using it, unqualified, on the left-hand side of some rule. Rules can only prove derived constants or values built from constructors. This rule causes an error:

```
1 :- V(_).
```

```
ex.4ml (1, 4): Syntax error - A rule cannot produce
the base constant 1
```

The full name of a derived constant c declared in domain D is $D.c$. This allows for (an optional) coding idiom where each derived constant is declared in exactly one place.

```
Declares hasCycle, because appears unqualified on left-hand side of the rule.
(Does not affect the provable values.)
hasCycle :- RelTrees.hasCycle.
First use of hasCycle; not a declaration because used in qualified form.
RelTrees.hasCycle :- anc(u, u).
Second use of hasCycle; this rule is redundant.
RelTrees.hasCycle :- anc(u, w), anc(w, u).
```

3.2 Rule Bodies

The body of a rule is a conjunction of constraints. Constraints can be either *matching constraints* or *interpreted predicates*. A matching constraint is satisfied if there is some substitution of the variables where the resulting value is provable. An interpreted predicate is satisfied if there is some substitution of the variables where a special predicate evaluates to true. Tables 5 and 6 list the forms of matching constraints and interpreted predicates. The order in which conjuncts appear is irrelevant; as with queries, all variables must be orientable.

To demonstrate interpreted predicates, consider that our *RelTrees* domain also allows a single node to have more than two left (or right) children.

```
model TwoLeftChildren of RelTrees
{
   v1 is V(1). v2 is V(2). v3 is V(3).
   Parent(v1, ROOT).
   Parent(v2, Context(LFT, v1)).
   Parent(v3, Context(LFT, v1)).
}
```

The following rule can be used to detect these anomalous graphs:

```
tooManyChildren :-
   Parent(x, Context(pos, parent)),
   Parent(y, Context(pos, parent)),
   x != y.
```

The disequality predicate (‘!=’) is used to detect if more than one distinct node has been assigned to the same position in the parent. Using the equality predicate (‘=’) and the selector operator (‘.’), the above rule can be rewritten as:

```
tooManyChildren :-
   px is Parent,
   py is Parent,
   px.chld != py.chld,
   px.cxt = py.cxt,
   px.cxt != ROOT.
```

The choice of whether to use the first or second rule is a matter of style. (Other equivalent rules can be written too.)

3.3 Interpreted Functions

FORMULA also provides many *interpreted functions*, such as $+$, $and(,)$, $toString()$, etc... An interpreted function can appear anywhere a t, t' or t_i appears in Tables 5 and 6. (See Section 5 for the full list of interpreted functions.) This constraint finds all the vertices whose child labels sum to the label of the parent.

```
SumToParent(v) :-
   Parent(x, Context(_, v)),
   Parent(y, Context(_, v)),
   x.lbl + y.lbl = v.lbl.
```

The interpreted function $+$ represents ordinary arithmetic addition, which is only defined for numeric values. Therefore *x.lbl* and *y.lbl* must be numeric values for the operation to be meaningful. These extra requirements are automatically added as *side-constraints* to the rule body. The complete rule constructed by the FORMULA compiler is:

```
SumToParent(v) :-
   Parent(x, Context(_, v)),
   Parent(y, Context(_, v)),
   x.lbl + y.lbl = v.lbl,
   xlbl = x.lbl, xlbl : Real,
   ylbl = y.lbl, ylbl : Real.
```

These side-constraints have an important consequence: A rule will never evaluate operations on badly-typed values. The side-constraints guarantee that the rule is not triggered for values where the operators are undefined.

To demonstrate this, create a new file containing the code in Example 5:

Example 5 (Pretty labeled trees).

```
domain PrettyRelTrees
{
   V       ::= new (lbl: Integer + String).
   Parent  ::= fun (chld: V => cxt: any {ROOT} + Context).
   Context ::= new (childPos: {LFT, RT}, prnt: V).

   SumToParent ::= (V).
   SumToParent(v) :-
      Parent(x, Context(_, v)),
      Parent(y, Context(_, v)),
      x.lbl + y.lbl = v.lbl.
}

model StringTree of PrettyRelTrees
{
   vA is V("a").
   vB is V("b").
   vC is V("c").

   Parent(vB, ROOT).
   Parent(vA, Context(LFT, vB)).
   Parent(vC, Context(RT, vB)).
}
```

In this domain vertices can have either integer labels or "prettier" string labels. The model *StringTree* contains a single tree with only string labels. Run the query:

```
[]> query StringTree SumToParent(_)
```

and notice that it evaluates to false. The query does not produce any errors or exceptions. The side-constraints created by + simply prevents the rule from being triggered by the string labeled vertices in *StringTree*.

3.4 Type Environments

In fact, FORMULA uses type inference to inform you about the values that may trigger a rule. Type:

```
[]> types PrettyRelTrees
```

and the following listing is returned:

```
+ Type environment at (8, 4)
  v: V(Integer)
  ~dc0: Parent(V(Integer), Context({RT} + {LFT}, V(Integer)))
  x: V(Integer)
  ~dc1: {RT} + {LFT}
  ~dc2: Parent(V(Integer), Context({RT} + {LFT}, V(Integer)))
  y: V(Integer)
  ~dc3: {RT} + {LFT}
  ~sv0: Integer
  ~sv1: Integer
  ~sv2: Integer
   + Type environment at (8, 22)
     v: V(Integer)
     ~dc0: Parent(V(Integer), Context({RT} + {LFT}, V(Integer)))
```

```
15:      x: V(Integer)
16:      ~dc1: {RT} + {LFT}
17:      ~dc2: Parent(V(Integer), Context({RT} + {LFT}, V(Integer)))
18:      y: V(Integer)
19:      ~dc3: {RT} + {LFT}
20:      ~sv0: Integer
21:      ~sv1: Integer
22:      ~sv2: Integer
```

This listing shows the inferred types for variables. Type inference is displayed as a sequence of nested *type environments*. The first type environment (line 1) lists the inferred types for variables in the head of the rule. The second nested type environment (line 12) lists the inferred types for the body of the rule. Had the rule contained several bodies via the semicolon operator, then there would be a distinct type environment for each body, and all of these environments would be nested under the head. The type of the head would then be the union of all the types in the bodies.

Line 2 shows that the variable v is guaranteed to be of type $V(Integer)$; it can never be a string labeled vertex. The variables named *~dc0*, *~dc1*, *~dc2*, *~dc3* are *don't care* variables. They were generated by the compiler wherever we preferred not to provide a variable name. This happened in four places in the body:

```
Parent(x, Context(_, v)), Parent(y, Context(_, v))
```

In the first matching constraint we did not provide a variable to bind the matching constraint (using the *is* operator). The compiler generates one called *~dc0* and its type is:

```
Parent(V(Integer), Context({RT} + {LFT}, V(Integer)))
```

This matching constraint is guaranteed to match only *Parent* values whose child is an integer-labeled vertex in the context of an integer-labeled parent. The variable *~dc1* occurs because we used the underscore operator ('_') in the first argument of *Context*. The variables *~dc2* and *~dc3* were analogously created for the second pattern. Finally, the variables *~sv0*, *~sv1*, and *~sv2* are compiler-generated *selector variables*. They stand for the selection of fields:

```
~sv0 = x.lbl, ~sv1 = y.lbl, ~sv2 = z.lbl
```

Examining the type environments is useful for understanding how constraints in the body interact to restrict the triggering of rules. FORMULA only generates an error if it is impossible to trigger a rule. For instance, change the declaration of V to:

```
V ::= new (lbl: String).
```

and reload the program. This will result in errors:

```
ex.4ml (8, 80): Argument 1 of function + is badly typed.
ex.4ml (8, 80): Argument 2 of function + is badly typed.
```

3.5 Set Comprehensions

Sometimes it is necessary to aggregate *all* the provable values of a given type into a single result. The rules we have shown so far cannot accomplish this task. Consider again the *IntGraphs* example (Example 1) and imagine trying to compute the *in-degree* of a vertex, i.e. the number of distinct edges coming into a vertex. One might be tempted to write rules such as:

```
indeg_atleast_2(v) :-
   v is V,
   e1 is E(_, v), e2 is E(_, v),
   e1 != e2.

indeg_atleast_3(v) :-
   v is V,
   e1 is E(_, v), e2 is E(_, v), e3 is E(_, v),
   e1 != e2, e1 != e3, e2 != e3.
```

These rules can only determine lower bounds on the in-degree by testing for vertices with at least k distinct incoming edges. Using this approach, we must write a rule for every possible in-degree that could be encountered. (We could never write all such rules.) Also, the rules would get larger; the rule computing *indeg_atleast_k* would contain $O(k^2)$ constraints. Also, even with these two rules, we still cannot write a rule that finds vertices whose in-degree is exactly two. A vertex v has a degree of exactly two if *indeg_atleast_2*(v) is provable and *indeg_atleast_3*(v) is *not* provable. But so far we cannot test if *indeg_atleast_3*(v) is *not* provable in the body of a rule.

Set comprehensions remedy this problem and fundamentally increase the expressive power of FORMULA. A set comprehension has the following form:

```
{ t1, ..., tn | body }
```

The body part can be legal rule body (e.g. it can contain the semicolon operator and nested comprehensions). The expressions $t_1, \ldots, t_n$ can be any legal combination of constants, constructors, variables, and selectors. A set comprehension means:

"Form a set S of values as follows: For every assignment of variables satisfying the body substitute these values in to each t_i and add each t_i to the set S.

Here is an example of using a set comprehension to compute the in-degree of an arbitrary vertex.

```
indeg(v, k) :- v is V, k = count({ e | e is E(_, v)}).
```

The *count*() operation is an interpreted function that takes a set comprehension and returns its size. The comprehension forms a set of all the edges with v in the destination position, and k is equal to the size of this set. Notice that the comprehension sees the variables declared outside of it, and so each choice of value for the variable v instantiates v inside the comprehension. However, variables introduced inside the comprehension do not escape. The variable e is scoped to the comprehension and other comprehensions cannot see it. Consider this rule:

```
q :- count({x | V(x), x > 0}) = count({x | V(x), x < 0}).
```

The first occurrence of x is lexically scoped to the first comprehension, and second occurrence is scoped to the second comprehension. This rule tests if the number of vertices with positive labels equals the number with negative labels. This rules *does not* constrain x to be both positive and negative (which is impossible).

Comprehensions can only be used in combination with certain interpreted predicates / interpreted functions. Variables can only be assigned values, so it is illegal to assign a set comprehension to a variable. The interpreted predicate *no* is used to test if a comprehension is empty, it is equivalent to the constraint:

```
count({ ... }) = 0
```

This rule computes all the *sources* in a graph, i.e. all the vertices with zero in-degree.

```
source(v) :- v is V, no { e | e is E(_, v) }.
```

The body of this comprehension consists of a single matching constraint. For the special case where *no* is applied to a comprehension with a single matching constraint, then only the body of the comprehension needs to be written.

```
source(v) :- v is V, no e is E(_, v).
```

Equivalently,

```
source(v) :- v is V, no E(_, v).
```

3.6 General Rules and Rule Heads

A general rule has for the form:

```
head1, ..., headm :- body1; ...; bodyn.
```

It is equivalent to the set of rules:

```
head1, ..., headm :- body1.
```

$\vdots$

```
head1, ..., headm :- bodyn.
```

A general rules proves all heads $head_1, \ldots, head_m$ for every satisfying assignment of the body. Each head must be formed from constants, constructors, and variables. A rule head must satisfy the following properties:

– Under all circumstances, a rule head must evaluate to a derived constant or a constructed value.
– Under all circumstances, a rule head must evaluate to a well-typed value.
– Every variable appearing in a rule head must appear at the top level of the body. (The variable cannot be introduced by a comprehension.)

All of these properties are checked at compile time and generate errors if violated. A rule violating the first property was demonstrated earlier (i.e. a rule that proved the constant 1). The second property is the most interesting. It requires the constraints in the body to prove that every possible assignment satisfying the body yields a well-typed head value. Imagine adding the following code to Example 5 where vertices can have integer or string labels:

```
1:    IntLabel ::= (Integer).
2:
3:    IntLabel(x) :- V(x).
4:    isIntLabel  :- IntLabel(x), V(x).
```

The rule in line 3 matches a vertex; the inferred type of x is $String + Integer$. However, the head *IntLabel*(x) requires x to be an integer. If the rule were triggered by a string-labeled vertex, then it would produce a badly-typed head value. This danger is detected by the FORMULA compiler:

```
ex.4ml (3, 4): Argument 1 of function IntLabel is unsafe.
     Some values of type String are not allowed here.
```

Contrast this with the rule in line 4. Here the conjuncts *IntLabel*(x) and $V(x)$ constrain x to be an integer. This rule is accepted by the compiler. Remember that the bodies must constrain variables enough so that the compiler can prove the heads are always well-typed. This architecture is designed to detect mistakes in rules at compile time.

There is one exception to the previous discussion. Selectors can be used in the head of a rule, in which case they are treated as if they occurred the body. This mean selectors appearing in the head will constrain the variables appearing in the body. This rule computes if one vertex is a child of another in a relational tree:

```
isChild(p.chld, p.cxt.prnt) :- p is Parent.
```

The occurrence of *p.cxt.prnt* constrains p to have the type $Parent(V, Context)$ even though no such constraint appears directly in the body. The rule the compiler produces is actually:

```
isChild(p.chld, p.cxt.prnt) :-
   p is Parent, _ = p.chld, _ = p.cxt.prnt.
```

Finally, every variable appearing in a rule head must also appear at the top level of a rule body. This rule is illegal:

```
isChild(p.chld, x) :- p is Parent.
```

because x does not appear in the body. This restriction prevents rules from proving an infinite number of facts, thereby preserving executability of rules. More subtly, this rule is also illegal:

```
isChild(x, y) :- no Parent(x, Context(_, y)).
```

Intuitively, this rule succeeds if there is no *Parent* value matching the constraint, so no assignments are available for the variables x and y in the head. A more general way to understand the problem is to re-introduce the lexical scoping braces:

```
isChild(x, y) :- no { p | p is Parent(x, Context(_, y)) }.
```

The variables x and y are not visible outside of the set comprehension, so they cannot be used in the head. Finally, a *fact* is a rule whose head contains no variables and whose body is empty. It is treated as a rule whose heads are always provable. A fact can be written as:

```
value1, ..., valuem.
```

3.7 Stratification and Termination

Rules are a form of executable logic. This means we can simultaneously treat them as a set of logical statements or as a kind of program, but both points-of-view should agree on what the rules mean. Obtaining this agreement becomes complicated without additional requirements on the structure of rules. The first major challenge arises because of set comprehensions. Consider these rules:

```
p :- no q.
q :- no p.
```

To execute these rules as program, FORMULA (1) chooses a rule, (2) computes all the new values it proves, and (3) repeats steps 1-2 until no new values are proved. If the first rule is executed first then p is proved because q is not, but then q cannot be proved. On the other hand, if the second rule is executed first then q is proved because p is not, but then p cannot be proved. Consequently, the answer to `query M p` depends on the order of execution.

There are two ways to reconcile this behavior. Either we can treat this behavior as correct, in which case the logical interpretation of rules must be generalized. Or, we can restrict the structure of rules to prevent this behavior all together. We choose the second approach, and prevent rules that would exhibit the behavior just described. One well-known restriction that eliminates this problem is called *stratification.*

Definition 31 (Stratification)**.** A FORMULA program is stratified if there is no set comprehension that examines values (indirectly) proved by the rule containing the comprehension.

In the previous example the first rule contains a set comprehension *no* $\{q \mid q\}$. And, q values are proved by the second rule, which examines p values (under a set comprehension of its own). Therefore the first set comprehension examines values that could be indirectly proved by the rule containing it. This is a sign that the order of execution could yield different outcomes. FORMULA produces the following error message for the previous program:

```
ex.4ml (1, 9): A set comprehension depends on itself.
               Listing dependency cycle 0...
ex.4ml (1, 9): A set comprehension depends on itself.
               Dependency cycle 0
ex.4ml (1, 4): A set comprehension depends on itself.
               Dependency cycle 0
ex.4ml (2, 9): A set comprehension depends on itself.
               Dependency cycle 0
ex.4ml (2, 4): A set comprehension depends on itself.
               Dependency cycle 0
```

The error messages list the locations of the rules and the set comprehensions that form a dependency cycle. Stratification is fully checked at compile time and unstratified programs are rejected.

Rules are executed so that all values are proved before a dependent set comprehension is executed. This strategy always computes a unique result, which coincides with the logical interpretation of rules. Users may write rules in any syntactic order, but they will always be executed to respect the dependencies of set comprehensions. For example:

```
smallInDegree :- no { v | indeg(v, k), k > 3 }.
indeg(v, k)   :- v is V, k = count({ e | e is E(_, v)}).
```

A graph has a “small in-degree” if no in-degree is greater than three. The first rule computes *smallInDegree* by comprehending over all the *indeg* values. Furthermore, the second rule computes *indeg* values by comprehending over all the edges. The order of execution will be: compute all the E values, and then compute all the *indeg* values, and then compute the *smallInDeg* value. It does not matter that *smallInDeg* appeared earlier in program text.

The second challenge with executable logic is *termination*.

Definition 32 (Termination)**.** A domain is terminating if the set of provable values is finite for every model of that domain.

A non-terminating domain may execute forever when evaluating a query. Currently FORMULA does not check for termination, so a user may write a non-terminating program and later find that query execution never stops. Theoretically termination cannot be checked with certainty for arbitrary programs, though many conservative analyses are possible and FORMULA may use some of them in the future. Here is a classic example of a non-terminating rule representing the *successor function* $s()$.

```
s(x) :- x = 0; x is s.
```

The value $s(0)$ is provable, and so the value $s(s(0))$ is provable, and so the value $s^n(0)$ is provable for every positive integer n. In fact, the successor function is one way to represent the natural numbers. Users should not try to axiomatize theories, such as the theory of natural numbers, using FORMULA. Instead, they should utilize the interpreted functions that already embed these theories into the FORMULA language.

3.8 Complex Conformance Constraints

We have shown how rules can compute properties of models. To create a conformance constraint use the following syntax.

```
conforms body.
```

Definition 33 (Model Conformance)**.** A model conforms to its domain if:

– Every assertion in M is constructed from new-kind constructors.
– Every value in M is well-typed.
– The body of every conformance constraint is satisfied for some substitution of the variables.

Internally, FORMULA creates a special derived constant *D.conforms* for each domain D. All conformance constraints must be provable for *D.conforms* to be provable. Additionally, conformance constraints are introduced for the relation / function constraints appearing in type declarations. Table 7 lists the predefined derived constants. Users may write rules referring to these constants, but it is illegal to add a rule that proves them. Only the compiler can add rules proving these constants. We now list the complete domains for directed acyclic graphs and relational trees.

Example 6 (Directed Acyclic Graphs).

```
domain DAGs
{
    New-kind constructors.
    V ::= new (lbl: Integer).
    E ::= new (src: V, dst: V).
    Derived-kind constructors.
    path ::= (V, V).
    Computation of transitive closure.
    path(u, w) :- E(u, w); E(u, v), path(v, w).
    Acyclicity constraint.
    conforms no path(u,u).
}
```

Table 7. Table of predefined derived constants

Predefined Derived Constants

Name	Meaning
`D.conforms`	Provable if all conformance constraints are satisfied.
`D.notRelational`	Provable if a provable value contains $f(\ldots, t_i, \ldots)$, f is relational on position i, $t_i = g(\ldots)$, and $g(\ldots)$ is not provable.
`D.notFunctional`	Provable if a constructor is declared to be a (partial) function, but its provable values map an element from the domain of the function to several distinct elements in the codomain.
`D.notTotal`	Provable if a constructor is declared to be a total function, but some element of its domain is not mapped to an element of its codomain.
`D.notInjective`	Provable if a constructor is declared to be a (partial) injection, but several elements of its domain are mapped to the same element in its codomain.
`D.notInvTotal`	Provable if a constructor is declared to be a (partial) surjection, but there is an element of its codomain for which no element of the domain is mapped.

Example 7 (Relational Trees).

```
domain RelTrees_Final
{
    New-kind constructors.
    V       ::= new (lbl: Integer + String).
    Parent  ::= fun (chld: V => cxt: any {ROOT} + Context).
    Context ::= new (childPos: {LFT, RT}, prnt: V).
    Derived-kind constructors.
    anc ::= (ancestor: V, descendant: V).
    Computation of ancestors.
    anc(u, w) :- Parent(w, Context(_, u)); anc(u, v), anc(v, w).
    Computation of too-many children.
    tooManyChildren :-
       Parent(x, Context(pos, parent)),
       Parent(y, Context(pos, parent)),
       x != y.
    Conformance constraints
    conforms no anc(u, u).
    conforms no tooManyChildren.
}
```

3.9 Extracting Proofs

Often times it is useful to know *why* a query evaluates to true. If a query evaluates to true, then a *proof tree* can be obtained showing how the rules prove the query. In a new file type the code from Example 6 along with the code below:

```
model LittleCycle of DAGs
{
   v1 is V(1).
   v2 is V(2).
   E(v1, v2).
   E(v2, v1).
}
```

Execute the query (and observe that it evaluates to true):

```
[]> query LittleCycle path(u, u)
```

Next type:

```
[]> proof 0
```

(0 is the id of the query task; type the particular id of your task.) The following output is displayed:

```
1: Truth value: true
2:
3: _Query_263ea486_4485_4bff_bda4_c99ea8f94c27.requires :- (2, 1)
4:   ~dc0 equals
5:    _Query_263ea486_4485_4bff_bda4_c99ea8f94c27.~requires0 :- (2, 1)
6:      ~dc0 equals
7:       path(V(2), V(2)) :- (6, 4)
8:         ~dc0 equals
9:          E(V(2), V(1)) :- (14, 4)
10:          .
11:         ~dc1 equals
12:          path(V(1), V(2)) :- (6, 4)
13:            ~dc0 equals
14:             E(V(1), V(2)) :- (13, 4)
15:             .
16:          .
17:       .
18:    .
19: .
20:
21: Press 0 to stop, or 1 to continue
```

Line 1 indicates that the query evaluated to true. The remaining lines show a nested hierarchy of rules along with the values of the matching constraints that triggered the rule. The rules in lines 3 and 5 where generated by the compiler to hold the body of the query expression; they can be ignored. Line 7 shows

Table 8. Table of domain symbol placement

Placement of Domain Symbols	
Kind of Declaration	Symbols and Placement
`Type ::= ...`	The symbol `Type` and the *type constant* `#Type` are placed in the root. If declaration is an n-ary constructor declaration then type constants `#Type[0]`, ..., `#Type[n-1]` are also placed in the root.
`{ ..., Cnst, ... }`	The new-kind user constant `Cnst` is placed in the root.
`DerCnst :- ...`	The derived-kind user constant `DerCnst` is placed in the namespace D.
`x`	A variable introduced in a rule is placed in the root namespace. Variables can be introduced independently by many rules; this does not cause a conflict.
`D.Constant`	A predefined symbol defined to be the union of all new-kind constants (including numerics and strings). Placed in the namespace D along with `D.#Constant`.
`D.Data`	A predefined symbol defined to be the union of all new-kind constants (including numerics and strings) and data constructors. Placed in the namespace D along with `D.#Data`.
`D.Any`	A predefined symbol defined to be the union of all types in the domain. Placed in the namespace D along with `D.#Any`.

that a loop $path(V(2), V(2))$ was proved using the rule at line 6 column 4 in the domain definition (i.e. the transitive closure rule). This rule used the proofs of $E(V(2), V(1))$ (line 9) and $path(V(1), V(2))$ (line 12). The proof of $E(V(2), V(1))$ comes directly from the model assertion located at (14, 4). The proof of $path(V(1), V(2))$ requires another invocation of the transitive closure rule using the model assertion at (13, 4).

Press 1 key to obtain another proof of the query. Press the 1 key again and FORMULA exits the display loop because there are no more proofs to show. In fact, this statement is not entirely true; there are infinitely many proof trees that prove the loop, but most of them depend on a subproof of a loop and are not interesting. For instance, once $path(V(2), V(2))$ is proved, the the transitive closure rule can be applied again to obtain another proof for $path(V(2), V(2))$. FORMULA only shows minimal proof trees, and ignores (or *cuts*) proofs containing a subproof of the property.

The proof command is a bit more general. It can take a value (without variables) and return a truth value and possibly a proof tree. Try:

```
[]> proof 0 path(V(1), V(2))
```

Three results are possible:

– The truth value can be *true* and trees are displayed.
– The truth value can be *false* and no trees are displayed.
– The truth value can be *unknown*. This occurs if FORMULA did not evaluate enough rules to decide if the value is provable.

When a query operation is executed FORMULA will decide which rules are relevant to the query. It will report the truth value of *unknown* if the proof command is called with a value whose proof might require unexecuted rules. (This can also occur if the query execution was terminated prematurely.)

4 Domain and Model Composition

Domains and models can be combined to build up more complicated modules. Domain composition allows type declarations, rules, and conformance constraints to be combined. Model composition allows sets of assertions and aliases to be combined.

4.1 Namespaces

To understand composition, we must first discuss how symbols are organized. Every symbol s declared in a module is placed into a *namespace* n. The complete name of a symbol is $n.s$. We hinted at the existence of namespaces when describing derived constants such as *RelTrees.hasCycle* or *DAGs.conforms*. They also appeared in the symbolic constants *Gex'.%v1* and*Gex'.%e_1_2*. The *root namespace* has no name at all. The complete name of a symbol s placed in the root namespace is simply s. When modules are composed their namespaces are merged. Composition fails if the combined modules declare a symbol with the same full name but in semantically different ways.

Every domain D starts with two namespaces: the root namespace and a namespace D, which is a child of the root. Whether a declaration places symbols in the root or the D namespace depends on the kind of declaration. Table 8 describes the introduction and placement of domain symbols. Note that *type constants* are used to support reflection, but we have not discussed them yet.

Load the file containing Example 7 (*RelTrees_Final*) and type:

```
[]> det RelTrees_Final
```

You will see the complete set of symbols and their placement into namespaces. (Additional compiler generated symbols are listed as well.)

```
Symbol table
      Space       |       Name       | Arity | Kind
------------------|------------------|-------|-------
                  |      Boolean     |   0   |  unn
                  |      Context     |   2   |  con
                  |       FALSE      |   0   | ncnst
                  |      Integer     |   0   |  unn
                  |        LFT       |   0   | ncnst
                  |      Natural     |   0   |  unn
                  |    NegInteger    |   0   |  unn
                  |      Parent      |   2   |  map
                  |    PosInteger    |   0   |  unn
                  |       ROOT       |   0   | ncnst
                  |        RT        |   0   | ncnst
                  |       Real       |   0   |  unn
                  |      String      |   0   |  unn
                  |       TRUE       |   0   | ncnst
                  |         V        |   1   |  con
                  |        anc       |   2   |  con
  RelTrees_Final  |        Any       |   0   |  unn
  RelTrees_Final  |     Constant     |   0   |  unn
  RelTrees_Final  |       Data       |   0   |  unn
  RelTrees_Final  |     conforms     |   0   | dcnst
  RelTrees_Final  |   notFunctional  |   0   | dcnst
  RelTrees_Final  |   notInjective   |   0   | dcnst
  RelTrees_Final  |    notInvTotal   |   0   | dcnst
  RelTrees_Final  |   notRelational  |   0   | dcnst
  RelTrees_Final  |     notTotal     |   0   | dcnst
  RelTrees_Final  | tooManyChildren  |   0   | dcnst
  RelTrees_Final  |    ~conforms0    |   0   | dcnst
  RelTrees_Final  |    ~conforms1    |   0   | dcnst

Type constants:  #Boolean #Context #Context[0] #Context[1]
                 #Integer #Natural #NegInteger #Parent #Parent[0]
                 #Parent[1] #PosInteger #Real #String #V #V[0]
                 #anc #anc[0] #anc[1] RelTrees_Final.#Any
                 RelTrees_Final.#Constant RelTrees_Final.#Data
Symbolic constants:
Rationals:
Strings:
Variables: parent pos u v w x y ~arg1 ~arg2 ~arg2'
```

A model M of domain D contains all the same definitions as D but adds an additional namespace called M under the root. All aliases are placed here as symbolic constants.

```
[]> det LittleCycle
```

```
Listing omitted
Symbolic constants:  LittleCycle.%v1 LittleCycle.%v2
Listing omitted
```

Altogether the *LittleCycle* model contains three namespaces: the root namespace and two sub-namespaces called *DAGs* and *LittleCycle*.

4.2 Domain Composition

Domains are composed by writing:

```
domain D includes D1, ..., Dn { ... }
```

Composition imports all the declarations of $D_1, \ldots, D_n$ into D. If a symbol is in namespace n in D_i, then it remains in namespace n in D. If this merging causes a symbol to receive contradictory declarations, then an error is reported. As a corollary, importing the same domain several times has no effect, because all declarations trivially agree. Here is an example of a problematic composition:

```
domain D1 {  q :- X = 0. }
domain D2 {  T ::= { X }. }
domain D includes D1, D2 { }
```

In domain D_1 the symbol X is a variable, but in domain D_2 it is a user constant. These declarations are incompatible and an error is returned:

```
ex.4ml (3, 23): The symbol X has multiple definitions.
```

Here is a more subtle example:

```
domain D1 {  List ::= new (Integer, any List + {NIL}). }
domain D2 {  List ::= new (Real, any List + {NIL}). }
domain D includes D1, D2 { }
```

Though both domains agree that $List(,)$ is a binary constructor; they disagree on the type constraints. However, this composition is legal:

```
domain D1
{  List ::= new (Integer, any List + {NIL}). }
domain D2
{
   List ::= new (NegInteger + {0} + PosInteger, any List + {NIL}).
}
domain D includes D1, D2
```

```
{
   List       ::= new (Integer, any ListOrNone).
   ListOrNone ::= List + {NIL}.
}
```

Though *List* has a syntactically different definitions in each domain, all domains semantically agree on the values accepted by the *List*(,) constructor.

The *includes* keyword merges domain declarations. However, it does not force the importing domain to satisfy all the domain constraints of the imported domains. In the previous examples D contains rules for computing D*.conforms*, D_1*.conforms*, and D_2*.conforms*. However, D*.conforms* need not reflect the conformance constraints of D_1 and D_2. (Relation / function constraints will be respected, because they occur on type declarations.) A composite domain automatically inherits conformance constraints if the *extends* keyword is used in place of *includes*:

```
domain D extends D1, ..., Dn { ... }
```

Consider that the set of all DAGs is a subset of the set of all directed graphs, and the set of trees is a subset of the set of DAGs. This chain of restrictions can be specified as follows:

Example 8 (Classes of graphs).

```
domain Digraphs
{
   V ::= new (lbl: Integer).
   E ::= new (src: V, dst: V).
}

domain DAGs extends Digraphs
{
   path ::= (V, V).
   path(u, w) :- E(u, w); E(u, v), path(v, w).
   conforms no path(u, u).
}

domain Trees extends DAGs
{
   conforms no { w | E(u, w), E(v, w), u != v }.
}
```

4.3 The Renaming Operator

Suppose we would like to build a domain representing two distinct graphs, i.e. with two distinct edge and vertex sets. One construction would be the following:

```
domain TwoDigraphs
{
   V1 ::= new (lbl: Integer).
   E1 ::= new (src: V1, dst: V1).

   V2 ::= new (lbl: Integer).
   E2 ::= new (src: V2, dst: V2).
}
```

The constructors $V_1()$ and $E_1(,)$ construct elements from the first graph and the constructors $V_2()$ and $E_2(,)$ construct elements from the second graph. A model would contain vertices and edges that could always be classified as belonging to either the first or second graph.

While this specification accomplishes the goal, it is not very satisfying. *TwoGraphs* contains two deep copies of the *Digraphs* domain, but the constructors have been renamed in an ad-hoc manner. FORMULA provides a methodological way to accomplish this same task: the *renaming operator* ('::').

```
rename::Module
```

The renaming operator creates a new module of the same kind with a namespace called *rename*. It copies all definitions from *module* under the *rename* namespace, and rewrites all copied declarations to reflect this renaming. The only symbols immune to this operation are new-kind constants, which remain at the root of the freshly created module. The renaming operator has many uses, though we only demonstrate a few uses now. The following domain describes the set of all pairs of isomorphic DAGs.

Example 9 (Isomorphic DAGs).

```
domain IsoDAGs extends Left::DAGs, Right::DAGs
{
   Iso ::= bij (Left.V => Right.V).

   conforms no { e | e is Left.E(u, w),
                     Iso(u, u'), Iso(w, w'),
                     no Right.E(u', w') }.
   conforms no { e | e is Right.E(u', w'),
                     Iso(u, u'), Iso(w, w'),
                     no Left.E(u, w) }.
}
```

This domain contains two copies of the *DAGs* domain under the renamings *Left* and *Right*. It contains two vertex constructors called *Left.V()* and *Right.V()* as well as two edge constructors called *Left.E(,)* and *Right.E(,)*. Because *IsoDAGs* extends the renamed domains, the left graph and the right graph must satisfy *Left.DAGs.conforms* and *Right.DAGs.conforms* respectively. In other words, the constraints on the renamed structures are preserved. Line 3 introduces a new bijection for witnessing the isomorphism between the left and the right vertices. Notice that the types *Left.V* and *Right.V* are immediately available for use. Finally, the two additional conformance constraints require the *Iso* bijection to relate the vertices such that *Iso* is a proper isomorphism. Here is a model of the domain:

```
model LittleIso of IsoDAGs
{
   v1L is Left.V(1).
   v2L is Left.V(2).
   v1R is Right.V(1).
   v2R is Right.V(2).

   Left.E(v1L, v2L).
   Right.E(v2R, v1R).

   Iso(v1L, v2R).
   Iso(v2L, v1R).
}
```

FORMULA provides several shortcuts to avoid fully qualifying symbols. First, a symbol only needs to be qualified until there is a unique shortest namespace containing the symbol. Suppose there are symbols $X.f()$ and $X.Y.Z.f()$. Then $f()$ will be resolved to $X.f()$ because this is the shortest namespace containing a symbol called $f()$. Also, $Y.f()$ will be resolved to $X.Y.Z.f()$ because this is the shortest namespace containing a qualifier Y and the symbol $f()$. Second, the resolved namespace of a constructor is applied to arguments of the constructor. If this resolution fails, then resolution restarts from the root namespace. Consider this example:

```
Left.E(V(1), V(2))
```

The outer constructor *Left.E*$(,)$ is resolved to be in the *Left* namespace. Next, the inner constructor $V()$ is encountered and name resolution looks for a unique shortest qualifier under the *Left* namespace. This succeeds and resolves as *Left.V*$()$. Thus, we have avoided writing the qualifier *Left* on the occurrences of $V()$. Of course, it is acceptable to fully qualify the inner constructors.

```
Left.E(Left.V(1), Left.V(2))
```

In this case, the symbol *Left.Left.V*$()$ does not exist and so name resolution restarts from the root and resolves to *Left.V*$()$.

5 Interpreted Functions

5.1 Arithmetic Functions and Identities

Table 9. Table of arithmetic functions (I)

Arithmetic Functions (I)

Syntax	Side Constraints	Result
`-x`	`x : Real`	$-x$
`x + y`	`x : Real, y : Real`	$x + y$
`x - y`	`x : Real, y : Real`	$x - y$
`x * y`	`x : Real, y : Real`	$x \cdot y$
`x / y`	`x : Real, y : Real, y != 0.`	$\frac{x}{y}$
`x % y`	`x : Real, y : Real, y != 0.`	$0 \leq r < \lvert y \rvert$, such that $\exists q \in \mathbb{Z}.\ x = q \cdot y + r$.
`count({...})`	–	The number of elements in $\{\ldots\}$.
`gcd(x, y)`	`x : Integer, y : Integer`	$\overset{def}{=} \begin{cases} \lvert x \rvert \text{ if } y = 0, \\ gcd(y, \lvert x \rvert \% \lvert y \rvert). \end{cases}$
`gcdAll(x, {...})`	–	The gcd of all integer elements, or x if there are no such elements.
`lcm(x, y)`	`x : Integer, y : Integer`	$\overset{def}{=} \begin{cases} 0 \text{ if } \lvert x \rvert + \lvert y \rvert = 0, \\ \lvert x \cdot y \rvert / gcd(x, y). \end{cases}$
`lcmAll(x, {...})`	–	The lcm of all integer elements, or x if there are no such elements.
`max(x, y)`	–	x if $x \geq y$; otherwise y.
`maxAll(x, {...})`	–	The largest element of $\{\ldots\}$ in the order of values; x if $\{\ldots\}$ is empty.

Table 10. Table of arithmetic functions (II)

Arithmetic Functions (II)

Syntax	Side Constraints	Result
`min(x, y)`	–	x if $x \leq y$; otherwise y.
`minAll(x, {...})`	–	The smallest element of $\{\ldots\}$ in the order of values; x if $\{\ldots\}$ is empty.
`prod(x, {...})`	–	$\overset{def}{=} \begin{cases} x \text{ if } \{\ldots\} \cap \mathbb{R} = \emptyset, \\ \prod\limits_{e \in \{\ldots\} \cap \mathbb{R}} e. \end{cases}$
`qtnt(x, y)`	`x : Real`, `y : Real`, `y != 0`.	$q \in \mathbb{Z}$, such that $\exists 0 \leq r < \lvert y \rvert .\ x = q \cdot y + r$.
`sign(x)`	`x : Real`	$\overset{def}{=} \begin{cases} -1 \text{ if } x < 0, \\ 0 \quad \text{if } x = 0, \\ 1 \quad \text{if } x > 0 \end{cases}$
`sum(x, {...})`	–	$\overset{def}{=} \begin{cases} x \text{ if } \{\ldots\} \cap \mathbb{R} = \emptyset, \\ \sum\limits_{e \in \{\ldots\} \cap \mathbb{R}} e. \end{cases}$

Table 11. Table of arithmetic identities (LHS-s are not built-in operations)

Arithmetic Identities

Left-Hand Side		Right-Hand Side
$abs(x)$	$=$	$max(x, -x)$.
$ceiling(x)$	$=$	$-qtnt(-x, 1)$.
$floor(x)$	$=$	$qtnt(x, 1)$.

5.2 Boolean Functions

Table 12. Table of Boolean functions

Boolean Functions		
Syntax	Side Constraints	Result
`and(x, y)`	`x : Boolean`, `y : Boolean`	$x \wedge y$.
`andAll(x, {...})`	–	$\overset{def}{=} \begin{cases} x \text{ if } \{\ldots\} \cap \mathbb{B} = \emptyset, \\ \bigwedge_{e \in \{\ldots\} \cap \mathbb{B}} e. \end{cases}$
`impl(x, y)`	`x : Boolean`, `y : Boolean`	$\neg x \vee y$.
`not(x)`	`x : Boolean`	$\neg x$.
`or(x, y)`	`x : Boolean`, `y : Boolean`	$x \vee y$.
`orAll(x, {...})`	–	$\overset{def}{=} \begin{cases} x \text{ if } \{\ldots\} \cap \mathbb{B} = \emptyset, \\ \bigvee_{e \in \{\ldots\} \cap \mathbb{B}} e. \end{cases}$

5.3 String Functions

In the table above, ϵ is the empty string and $s[i]$ is the single-character string at position i in s, for $0 \leq i < strLength(s)$.

Table 13. Table of string functions

String Operations

Syntax	Side Constraints	Result
`isSubstring(x, y)`	`x : String`, `y : String`.	`TRUE` if x is a substring of y; `FALSE` otherwise. The empty string is only a substring of itself.
`strAfter(x, y)`	`x : String`, `y : Natural`.	Returns the largest substring starting at position y, or ϵ if $y \geq strLength(x)$.
`strBefore(x, y)`	`x : String`, `y : Natural`.	Returns the largest substring ending before position y, or ϵ if $y = 0$.
`strFind(x, y, z)`	`x : String`, `y : String`.	Returns the index of the first occurrence of x in y; z if y never appears.
`strFindAll(x, y, z, w)`	x is a w-terminated natural-list type constant `#F`. `y : String`, `z : String`.	Returns a w-terminated F'-list of all the indices where y occurs in z; w if it never occurs.
`strGetAt(x, y)`	`x : String`, `y : Natural`.	$\overset{def}{=} \begin{cases} x[y] \text{ if } y < strLength(x), \\ \epsilon \text{ otherwise.} \end{cases}$.
`strJoin(x, y)`	`x : String`, `y : String`.	$\overset{def}{=} \begin{cases} y \text{ if } x = \epsilon, \\ x \text{ if } y = \epsilon, \\ xy \text{ otherwise.} \end{cases}$.
`strLength(x)`	`x : String`.	Returns the length of x.
`strLower(x)`	`x : String`.	Returns the all-lower-case version of x.
`strReverse(x)`	`x : String`.	Returns the reverse of x.
`strUpper(x)`	`x : String`.	Returns the all-upper-case version of x.

5.4 List Functions

A *list constructor* is a constructor `F ::= (T0, T1)` such that `F` is a subtype of `T1`. A *T-list constructor* is a list constructor such that `T` is a subtype of `T0`. A *list type constant* `#F` is a type constant such that `F` is a list constructor. A list is *flat* if it has n elements placed as follows:

$$F(t_0, F(t_1, \ldots, F(t_{n-1}, w) \ldots))$$

The value w is called the *terminator*. In the table below all operations, except for `isSubTerm` and `lstFlatten`, assume flat lists.

Table 14. Table of list functions

List Functions		
Syntax	Side Constraints	Result
`isSubterm(x, y)`	–	`TRUE` if x is a subterm of y; `FALSE` otherwise. .
`lstAfter(x, y, z, w)`	x is a w-terminated list type constant `#F`, `z : Natural`.	y if $y \neq F(\ldots)$; w if $z \geq lstLength(y)$; a w-terminated F-list of all the elements at and after z.
`lstBefore(x, y, z, w)`	x is a w-terminated list type constant `#F`, `z : Natural`.	y if $y \neq F(\ldots)$; w if $z \leq 0$; a w-terminated F-list of all the elements before z.
`lstFind(x, y, z, w)`	x is a list type constant `#F`.	The first place where z occurs in the F-list y; w if z never occurs.
`lstFindAll(x, x', y, z, w)`	x is a list type constant `#F`, x' is a w-terminated natural-list type constant `#F'`.	Returns a w-terminated F'-list of all the indices where z occurs in y; w if it never occurs.
`lstFlatten(x, y, w)`	x is a w-terminated list type constant `#F`.	Converts y into w-terminated flat form if $y = F(\ldots)$; y otherwise.
`lstGetAt(x, y, z)`	`z : Natural`, `z < lstLength(x, y)`.	$\overset{def}{=} \begin{cases} h \text{ if } lstGetAt(x, F(h,t), 0), \\ lstGetAt(x, t, z-1). \end{cases}$
`lstLength(x, y)`	x is a list type constant `#F`.	$\overset{def}{=} \begin{cases} 0 \text{ if } y \neq F(h,t), \\ 1 + lstLength(t). \end{cases}$
`lstReverse(x, y)`	x is a list type constant `#F`.	y if $y \neq F(\ldots)$; otherwise reverses the list reusing the same terminator

5.5 Coercion Functions

In the table to follow, a *list type constant* `#F` is a type constant such that `F ::= (T1, T2)` and `F <: T2`.

Table 15. Table of coercion functions

Coercion Functions

Syntax	Side Constraints	Result
`toList(x, y, {...})`	x is a list type constant, y : T2.	$F(t_1, \ldots, F(t_n, y) \ldots)$ where t_i are the sorted elements of $\{\ldots\}$ accepted by $T1$. Or y if no element is accepted.
`toNatural(x)`	–	Returns a unique natural for the value x.
`toString(x)`	–	Returns a unique string for the value x.
`toSymbol(x)`	–	Returns x if x is a constant; `#F` for $x = F(\ldots)$.

5.6 Reflection Functions

Table 16. Table of reflection functions

Reflection Functions

Syntax	Side Constraints	Result
`rflGetArgType(x, y)`	x is a type constant `#F`, `y : Natural`, `y < rflGetArity(x)`.	Returns `#F[y]`, where `F ::= (T0,...,T`$_{n-1}$`)`
`rflGetArity(x)`	x is a type constant `#X`.	Returns n if `X ::= (T0,...,T`$_{n-1}$`)`; 0 otherwise.
`rflIsMember(x, y)`	y is a type constant `#Y`.	`TRUE` if x is a member of Y; `FALSE` otherwise.
`rflIsSubtype(x, y)`	x is a type constant `#X`, y is a type constant `#Y`.	`TRUE` if X is a subtype of Y; `FALSE` otherwise.

Formal Modelling, Analysis and Verification of Hybrid Systems

Naijun Zhan, Shuling Wang, and Hengjun Zhao

State Key Lab. of Comput. Sci., Inst. of Software, Chinese Academy of Sciences

Abstract. Hybrid systems is a mathematical model of embedded systems, and has been widely used in the design of complex embedded systems. In this chapter, we will introduce our systematic approach to formal modelling, analysis and verification of hybrid systems. In our framework, a hybrid system is modelled using Hybird CSP (HCSP), and specified and reasoned about by Hybrid Hoare Logic (HHL), which is an extension of Hoare logic to hybrid systems. For deductive verification of hybrid systems, a complete approach to generating polynomial invariants for polynomial hybrid systems is proposed; meanwhile, a theorem prover for HHL that can provide tool support for the verification has been implemented. We give some case studies from real world, for instance, Chinese High-Speed Train Control System at Level 3 (CTCS-3). In addition, based on our invariant generation approach, we consider how to synthesize a switching logic for a considered hybrid system by reduction to constraint solving, to meet a given safety, liveness, optimality requirement, or any of their combinations. We also discuss other issues of hybrid systems, e.g., stability analysis.

Keywords: Hybrid systems, Hybrid CSP, Hybrid Hoare Logic, Invariant, Theorem proving.

1 Introduction

Our modern life increasingly depends on embedded systems. How to develop correct complex embedded systems is a grand challenge for computer science and control theory. The model-based method is thought to be an effective method to designing complex embedded systems. Using this approach at the very beginning, an abstract model of the system to be developed with precise mathematical semantics is defined. Extensive analysis and verification on the abstract model are then committed so that errors can be identified and corrected at the very early stage. Then, a higher-level abstract model is refined to a lower-level abstract model, even to source code, step by step, using model-transformation techniques.

Hybrid systems, combining formal models for discrete *reactive systems* and continuous models for *dynamical systems* [1,59], is a mathematical model of embedded systems. There are hugely numerous work that have been done related to hybrid systems. Please refer to [4,42] for a survey. Modeling discrete components by finite automata, and attaching state-dependent ordinary differential equations to the discrete states in order to capture the impact of the discrete component on the continuous environment, yields hybrid automata [1], which are by far the most widely used model of hybrid systems in academia. Hybrid automata are, however, analogs of state machine, with little support for structured description, and consequently, a number of formalisms have been proposed to facilitate modular descriptions of complex systems. These include modeling environments such as SHIFT [28] and PTOLEMY [31] for hierarchical specification of hybrid behavior; models such as hybrid I/O automata [58], hybrid modules [2], and CHARON [9], for compositional treatment of concurrent hybrid behavior; Hybrid CSP (HCSP) [37,98] for process algebra based specification and verification of hybrid behavior; and differential dynamic logic [65,66] and hybrid Hoare logic (HHL) [55,86] for logic-based specification and compositional analysis of hybrid behavior. Industrial variants include the Simulink/Stateflow environment, which does, however, lack a uniquely defined —depending on the use case, there are semantical variations— and comprehensive formal semantics.

With most hybrid system models being rooted in automata-based models, their pertinent verification techniques have accordingly adopted the automaton-based approach to verification in the past, mainly being based on directly computing exact representations or safe approximations of reachable state sets. For example, based on model-checking [22,71], the reachability problems of some simple hybrid systems, like timed automata [8], multirate automata [1], initialized rectangular automata [70,41], and so on, have been solved; based on the decision procedure of Tarski algebra [83], in [53] methods for computing reachable sets for three classes of special linear hybrid systems were investigated. Due to infiniteness of the underlying state spaces, symbolic representations, often paired with safe approximation within a computationally reasonably efficient symbolic representation, and abstraction techniques are generally applied in reachable set computation. For example, the tool HYTECH [3] was the first model checker to implement exact symbolic reachability analysis of linear hybrid automata[1] by using polyhedra-based technique; while the tools CHECKMATE [20] and **d/dt** [11] compute over-approximations of reachable sets of linear hybrid systems[1] using polyhedral representations; related techniques use lazy theorem proving for analyzing bounded reachability problems of linear [12] or non-linear [34] hybrid automata. Furthermore, discretization of continuous dynamics based on gridding or predicate abstraction has been extended and adopted for hybrid systems [40,7,21,73,25].

To deal with more complicated systems, recently, a deductive method for the verification of hybrid systems has been established and successfully applied in practice [65,66]. This method can be seen as a generalization of the so-called Floyd-Hoare-Naur inductive assertion method [32,43,62]. The inductive assertion

[1] With hybrid systems being an interdisciplinary domain bridging control theory and computer science, terminology often is subtle due to different roots of naming conventions. A *linear hybrid automaton* is a system featuring (piecewise) constant differential inclusions while a *linear hybrid system* features linear or often even affine differential equations.

method is thought to be the dominant method for the verification of sequential programs. To generalize the inductive method to hybrid systems, a modeling language with compositionality for hybrid systems and a Hoare-style logic for the language with the ability of dealing with continuous dynamics are prerequisites. For example, a differential-algebraic dynamic logic for hybrid programs [64] was invented by extending dynamic logic with continuous statements. Recently, we [55] had another effort by extending Hoare logic to hybrid systems modeled by HCSP [37,98] for the same purpose.

The concept of *invariant* is at the core of deductive methods. An *invariant* of a hybrid system is a property ϕ that holds in all the reachable states of the system. An *inductive invariant* of a hybrid system is an assertion ϕ that holds at the initial states of the system, and preserved by all discrete and continuous dynamics. In fact, any inductive invariant is also an invariant, but the inverse is not true in general. The problem of (inductive) invariant generation has received wide attention in the analysis and verification of programs [13,23,76,48] and hybrid systems [75,69,65,81,55]. Many properties of hybrid systems like safety, stability, liveness etc., can be characterized and inferred via invariants without solving differential equations, while differential equations have to be exactly solved or approximated in the methods based on directly computing reachable sets.

The key issue in generating inductive invariants of a hybrid system is to deal with continuous dynamics, i.e. to generate so-called *continuous invariant* (CI) of the continuous dynamics at each mode of the system. A method based on constraint solving was proposed in [75] for generating CIs containing a single polynomial equation. This method was generalized in [74] to construct CIs containing infinitely many polynomial equations, i.e. the so-called *invariant ideal.* The basic idea of these methods is to reduce the CI generation problem to a constraint solving problem using techniques from the theory of ideals over polynomial rings. For the polynomial inequality case, it was considered in [69,67] how to generate CIs containing one polynomial inequality. The basic idea of their method is to utilize a certain function, called a *barrier certificate*, to enclose the invariant. With some stronger constraints, generation of more general CIs was considered in [65], wherein the CIs are Boolean combinations of polynomial equations and inequalities. By restricting the invariant sets to have smooth boundaries, a sound but incomplete method for constructing invariants involving non-strict polynomial inequalities was proposed in [81,80]. While in [56], we presented a relatively complete method for generating *semi-algebraic invariants* (SAIs) for polynomial continuous dynamical systems by employing higher-order Lie derivatives and the theory of polynomial ideal.

As a complementation of verification, synthesis focuses on designing a controller that controls the underlying subsystems so that the whole system is guaranteed to satisfy the given requirement, that may be safety, liveness (e.g. reachability to a given set of states), optimality criterion, or a desired combination of them. Numerous work have been done on controller synthesis for safety and/or reachability requirements. For example, in [10,85], a general framework relying on *backward reachable set* computation and *fixed point iteration* was proposed, for synthesizing controllers for hybrid automata to meet a given safety requirement; while in [79], a symbolic approach based on templates and constraint solving to the same problem was proposed, and in [82], the symbolic approach is extended to meet both safety and reachability requirements. Compared with controller synthesis for safety, the optimal controller synthesis problem is more involved, also quite important in the design of hybrid systems. In the literature, few work has been done on the problem. Larsen et al proposed an approach based on energy automata and model-checking [18], while Jha, Seshia and Tiwari gave a solution to the problem using unconstrained numerical optimization and machine learning [44]. In [94], we proposed a "hybrid" approach for synthesizing optimal controllers of hybrid systems subject to safety requirements. The basic idea is as follows. Firstly, we reduce optimal controller synthesis subject to safety requirements to quantifier elimination (QE for short). Secondly, in order to make our approach scalable, we discuss how to combine QE with numerical computation, but at the same time, keep arising errors due to discretization manageable and within bounds. A major advantage of our approach is not only that it avoids errors due to numerical computation, but it also gives a better optimal controller.

All the aforementioned verification or synthesis approaches aim at showing or avoiding unreachability of undesirable states, i.e. total absence of undesirable behavior. In realistic applications, this often is an overly ambitious goal, being economically unattainable or even technically impossible to achieve due to uncontrollable environmental influences, unavoidable manufacturing tolerances, component breakdown, etc. Therefore, the existing, qualitative safety analysis methods for hybrid systems have to be complemented by quantitative methods, quantifying the likelihood of residual error or related performance figures (MTBF, MTTF, etc.) in systems subject to uncertain, stochastic behavior (both in the embedded system and its environment) as well as noise. It is therefore necessary to address such stochastic issues in the model of embedded systems, i.e. hybrid systems, by adding models of stochastic behavior to the modeling language and corresponding analysis techniques to the verification. Some first attempts on introducing probability and stochasticity in hybrid models have been pursued, e.g. [45,46,34,33], yet expressiveness of the models and scalability of the analysis tools remain pressing issues.

1.1 Synopsis

In Sec. 2, some basic notions, notations and mathematical foundations that will be used later are provided.

In Sec. 3, we introduce our approach for generating semi-algebraic invariants for polynomial continuous dynamical systems and its extension to hybrid systems. This is the first relatively complete approach for discovering polynomial invariants for these systems in the literature. This section is mainly based on our previous joint work with Liu reported in [56,54].

In Sec. 4, we first introduce how to synthesize switching controllers for hybrid systems subject to safety requirement based on continuous invariant generation reported in Sec. 3. To improve the efficiency, qualitative analysis [47] is adopted. This part is based on our recent joint work with Kapur [49]. Then, we consider optimal controller synthesis problems of hybrid systems by reducing to constraint solving, which is based on a joint work with Kapur and Larsen [94].

In Sec. 5, we introduce Hybrid CSP due to He, Zhou et al [37,98], which is an extension of CSP for hybrid systems. Here, we define a formal operational semantics for HCSP, which has been implemented in the HHL prover introduced later.

In Sec. 6, we introduce a specification logic for hybrid systems, called Hybrid Hoare Logic, which is achieved by combining Hoare logic with Duration Calculus (DC) [97,96]. The presentation is based on our previous work [55].

In Sec. 7, we introduce a proof assistant of HHL in Isabelle/HOL, which is based on our recent joint work with Zou et al [99].

In Sec. 8, we present a case study from a real world on Chinese high-speed train control system by using HCSP and HHL and the tool, based on the recent joint work [99].

In Sec. 9, we discuss other issues related to hybrid systems, mainly focusing on stability analysis of continuous dynamical systems based on a joint work with Liu [57].

Finally, we conclude this tutorial by Sec. 10 with some discussions of future work.

2 Preliminaries

In this section, we define the basic notions and notations that will be used in the rest of this tutorial. We also give an elementary description of several relevant mathematical theories fundamental to the understanding of this tutorial. For a comprehensive introduction of these theories the readers may refer to the cited literatures.

Throughout this tutorial, we use $\mathbb{N}, \mathbb{Q}, \mathbb{R}$ to denote the set of *natural*, *rational* and *real* numbers respectively. Given a set A, the Cartesian product of its n duplicates is denoted by A^n; for instance, $\mathbb{R}^n$ stands for the n-dimensional Euclidean space. A vector element $(a_1, a_2, \ldots, a_n) \in A^n$ is usually abbreviated by a boldface letter $\mathbf{a}$ when its dimension is clear from the context.

2.1 Continuous Dynamical Systems

We introduce some basic theories of continuous dynamical systems here. For details please refer to [50,84].

Typically, a continuous dynamical systems (CDS for short) is modeled by first-order autonomous ordinary differential equations

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}) \ , \tag{1}$$

where $\mathbf{x} \in \mathbb{R}^n$ and $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^n$ is a vector function, called a *vector field* in $\mathbb{R}^n$.

If $\mathbf{f}$ in (1) satisfies the *local Lipschitz condition*, then given $\mathbf{x}_0 \in \mathbb{R}^n$, there exists a unique *differentiable* vector function $\mathbf{x}(\mathbf{x}_0; t) : (a, b) \to \mathbb{R}^n$, where (a, b) is an open interval containing 0, such that $\mathbf{x}(\mathbf{x}_0; 0) = \mathbf{x}_0$ and the derivative of $\mathbf{x}(\mathbf{x}_0; t)$ w.r.t. t satisfies

$$\forall t \in (a, b).\ \frac{\mathrm{d}\mathbf{x}(\mathbf{x}_0; t)}{\mathrm{d}t} = \mathbf{f}(\mathbf{x}(\mathbf{x}_0; t)) \ .$$

Such $\mathbf{x}(\mathbf{x}_0; t)$ is called the solution to (1) with initial value $\mathbf{x}_0$.

If for any $\mathbf{x}_0 \in \mathbb{R}^n$, there is a solution $\mathbf{x}(\mathbf{x}_0; t)$ to (1) that exists for all time $t \in \mathbb{R}$, then the vector field $\mathbf{f}$ is called *complete*. A *globally Lipschitz continuous* vector field $\mathbf{f}$ guarantees the existence, uniqueness and completeness of solutions to (1).

If $\mathbf{f}$ is *analytic* at $\mathbf{x}_0 \in \mathbb{R}^n$, i.e. $\mathbf{f}$ is given by a convergent power series in a neighborhood of $\mathbf{x}_0$, then there exists a unique *analytic* solution $\mathbf{x}(\mathbf{x}_0; t)$ to (1) defined in a neighborhood of 0.

According to the evolution direction w.r.t. time, the solutions to (1) induce two sorts of geometrical curves as follows.

Definition 1. *Suppose* $\mathbf{x}(\mathbf{x}_0; t)$ *is the solution to (1) with initial value* $\mathbf{x}_0$. *Then*

– $\mathbf{x}(\mathbf{x}_0; t)$ *with* $t \geq 0$ *is called the* **trajectory** *of* $\mathbf{f}$ *starting from* $\mathbf{x}_0$*;*
– $\mathbf{x}(\mathbf{x}_0; -t)$ *with* $t \geq 0$ *is called the* **inverse trajectory** *of* $\mathbf{f}$ *starting from* $\mathbf{x}_0$*, where* $\mathbf{x}(\mathbf{x}_0; -t)$ *is obtained by substituting* $-t$ *for* t *in* $\mathbf{x}(\mathbf{x}_0; t)$*.*

When $\mathbf{x}_0$ is clear from the context, we write $\mathbf{x}(\mathbf{x}_0; t)$ and $\mathbf{x}(\mathbf{x}_0; -t)$ as $\mathbf{x}(t)$ and $\mathbf{x}(-t)$ for brevity.

The notion of *Lie derivative* is important for the study of CDSs and plays a central role in several subsequent sections of this tutorial. Let $\sigma(\mathbf{x}) : \mathbb{R}^n \to \mathbb{R}$ be a scalar function and $\mathbf{f}$ be a vector field in $\mathbb{R}^n$. Suppose both σ and $\mathbf{f}$ are *smooth functions*, i.e. differentiable in $\mathbf{x}$ at any order $k \in \mathbb{N}$. Then we can inductively define the *Lie derivatives* of σ along $\mathbf{f}$, i.e. $L_{\mathbf{f}}^k \sigma : \mathbb{R}^n \to \mathbb{R}$ for $k \in \mathbb{N}$, as follows:

– $L_{\mathbf{f}}^0 \sigma(\mathbf{x}) = \sigma(\mathbf{x})$,
– $L_{\mathbf{f}}^k \sigma(\mathbf{x}) = \big(\nabla L_{\mathbf{f}}^{k-1} \sigma(\mathbf{x}), \mathbf{f}(\mathbf{x})\big)$, for $k > 0$,

where ∇ stands for the *gradient* operator, i.e. for any differentiable function $\varrho(\mathbf{x}) : \mathbb{R}^n \to \mathbb{R}$,

$$\nabla \varrho(\mathbf{x}) \mathrel{\widehat{=}} \left(\frac{\partial \varrho(\mathbf{x})}{\partial x_1}, \frac{\partial \varrho(\mathbf{x})}{\partial x_2}, \ldots, \frac{\partial \varrho(\mathbf{x})}{\partial x_n} \right) ,$$

and $(\cdot, \cdot)$ is the *inner product* of two vectors, i.e. $(\mathbf{a}, \mathbf{b}) = \sum_{i=1}^n a_i b_i$ for $\mathbf{a} = (a_1, \ldots, a_n)$ and $\mathbf{b} = (b_1, \ldots, b_n)$.

2.2 Hybrid Systems

Hybrid systems are those systems that exhibit both continuous evolutions and discrete transitions between different modes. A widely adopted model of hybrid systems is *hybrid automata* [5,63,39], the extension of finite automata with continuous components. In this tutorial, the discussion of invariant generation and controller synthesis of hybrid systems will be cast in the setting of hybrid automata. A separate section (Sec. 5) of this tutorial will be devoted to a compositional language named HCSP, which is more suitable for modelling complex hybrid systems. The formal definition of hybrid automata in the literature differs slightly from each other. Here the presentation is based on [85] and [75].

Definition 2 (Hybrid Automaton). *A hybrid automaton (HA) is a system* $\mathcal{H} \mathrel{\widehat{=}} (Q, X, f, D, E, G, R, \Xi)$, *where*

- $Q = \{q_1, \ldots, q_m\}$ *is a finite set of discrete states (or modes);*
- $X = \{x_1, \ldots, x_n\}$ *is a finite set of continuous state variables, with* $\mathbf{x} = (x_1, \ldots, x_n)$ *ranging over* $\mathbb{R}^n$*;*
- $f : Q \to (\mathbb{R}^n \to \mathbb{R}^n)$ *assigns to each mode* $q \in Q$ *a locally Lipschitz continuous vector field* $\mathbf{f}_q$*;*
- D *assigns to each mode* $q \in Q$ *a mode domain* $D_q \subseteq \mathbb{R}^n$*;*
- $E \subseteq Q \times Q$ *is a finite set of discrete transitions;*
- G *assigns to each transition* $e \in E$ *a switching guard* $G_e \subseteq \mathbb{R}^n$*;*
- R *assigns to each transition* $e \in E$ *a reset function* $R_e \colon \mathbb{R}^n \to \mathbb{R}^n$*;*
- Ξ *assigns to each* $q \in Q$ *a set of initial states* $\Xi_q \subseteq \mathbb{R}^n$.

The state space of $\mathcal{H}$ is $\mathbb{H} \mathrel{\widehat{=}} Q \times \mathbb{R}^n$, the domain of $\mathcal{H}$ is $D_{\mathcal{H}} \mathrel{\widehat{=}} \bigcup_{q\in Q}(\{q\} \times D_q)$, and the set of all initial states is denoted by $\Xi_{\mathcal{H}} \mathrel{\widehat{=}} \bigcup_{q\in Q}(\{q\} \times \Xi_q)$. The semantics of $\mathcal{H}$ can be characterized by the set of *hybrid trajectories* accepted by $\mathcal{H}$ or the *reachable set* of $\mathcal{H}$.

Definition 3 (Hybrid Time Set). *A hybrid time set is a sequence of intervals* $\tau = \{I_i\}_{i=0}^{N}$ *(*N *can be* ∞*) such that:*

- $I_i = [\tau_i, \tau_i']$ *with* $\tau_i \le \tau_i' = \tau_{i+1}$ *for all* $i < N$*;*
- *if* $N < \infty$*, then* $I_N = [\tau_N, \tau_N'\rangle$ *is a right-closed or right-open nonempty interval (*τ_N' *may be* ∞*);*
- $\tau_0 = 0$.

Given a hybrid time set, let $\langle\tau\rangle = N$ and $\|\tau\| = \sum_{i=0}^{N}(\tau_i' - \tau_i)$. Then τ is called *infinite* if $\langle\tau\rangle = \infty$ or $\|\tau\| = \infty$, and *zeno* if $\langle\tau\rangle = \infty$ but $\|\tau\| < \infty$.

Definition 4 (Hybrid Trajectory). *A hybrid trajectory of* $\mathcal{H}$ *starting from an initial point* $(q_0, \mathbf{x}_0) \in \Xi_{\mathcal{H}}$ *is a triple* $\omega = (\tau, \alpha, \beta)$*, where* $\tau = \{I_i\}_{i=0}^{N}$ *is a hybrid time set, and* $\alpha = \{\alpha_i : I_i \to Q\}_{i=0}^{N}$ *and* $\beta = \{\beta_i : I_i \to \mathbb{R}^n\}_{i=0}^{N}$ *are two sequences of functions satisfying:*

1. *Initial condition:* $\alpha_0[0] = q_0$ *and* $\beta_0[0] = \mathbf{x}_0$*;*
2. *Discrete transition: for all* $i < \langle\tau\rangle$, $e = \big(\alpha_i(\tau_i'), \alpha_{i+1}(\tau_{i+1})\big) \in E$, $\beta_i(\tau_i') \in G_e$ *and* $\beta_{i+1}(\tau_{i+1}) = R_e(\beta_i(\tau_i'))$*;*
3. *Continuous evolution: for all* $i \le \langle\tau\rangle$ *with* $\tau_i < \tau_i'$*, if* $q = \alpha_i(\tau_i)$*, then*
   (1) for all $t \in I_i$, $\alpha_i(t) = q$,
   (2) $\beta_i(t)$ *is the solution to the differential equation* $\dot{\mathbf{x}} = \mathbf{f}_q(\mathbf{x})$ *over* I_i *with initial value* $\beta_i(\tau_i)$*, and*
   (3) for all $t \in [\tau_i, \tau_i')$, $\beta_i(t) \in D_q$.

The set of trajectories starting from an initial state $(q_0, \mathbf{x}_0)$ of $\mathcal{H}$ is denoted by $\mathcal{T}r(\mathcal{H})(q_0, \mathbf{x}_0)$, and the set of all trajectories of $\mathcal{H}$ by $\mathcal{T}r(\mathcal{H})$.

A hybrid trajectory $\omega = (\tau, \alpha, \beta)$ is called *infinite* or *zeno*, if τ is infinite or zeno respectively. An HA $\mathcal{H}$ is called *non-blocking* if for any $(q_0, \mathbf{x}_0) \in \Xi_{\mathcal{H}}$ there exists an infinite trajectory in $\mathcal{T}r(\mathcal{H})(q_0, \mathbf{x}_0)$, and *blocking* otherwise; $\mathcal{H}$ is called *non-zeno* if there exists no zeno trajectory in $\mathcal{T}r(\mathcal{H})$, and *zeno* otherwise.

Another way to interpret hybrid automata is using reachability relation.

Definition 5 (Reachable Set). *Given an HA $\mathcal{H}$, the reachable set of $\mathcal{H}$, denoted by $\mathcal{R}_{\mathcal{H}}$, consists of those $(q, \mathbf{x})$ for which there exists a finite sequence*

$$(q_0, \mathbf{x}_0), (q_1, \mathbf{x}_1), \ldots, (q_l, \mathbf{x}_l)$$

such that $(q_0, \mathbf{x}_0) \in \Xi_{\mathcal{H}}$, $(q_l, \mathbf{x}_l) = (q, \mathbf{x})$, and for any $0 \le i \le l - 1$, one of the following two conditions holds:

- *(Discrete Jump): $e = (q_i, q_{i+1}) \in E$, $\mathbf{x}_i \in G_e$ and $\mathbf{x}_{i+1} = R_e(\mathbf{x}_i)$; or*
- *(Continuous Evolution): $q_i = q_{i+1}$, and there exists a $\delta \ge 0$ s.t. the solution $\mathbf{x}(\mathbf{x}_i; t)$ to $\dot{\mathbf{x}} = \mathbf{f}_{q_i}$ satisfies*
  - *$\mathbf{x}(\mathbf{x}_i; t) \in D_{q_i}$ for all $t \in [0, \delta]$; and*
  - *$\mathbf{x}(\mathbf{x}_i; \delta) = \mathbf{x}_{i+1}$.*

Note that there is a subtle difference between Definition 4 and 5 in how to treat a continuous state $\mathbf{x}$ which terminates a piece of continuous evolution and evokes a discrete jump. Definition 4 is less restrictive because such $\mathbf{x}$ is not required to be inside the mode domain before jump happens. Nevertheless, if all mode domains are assumed to be *closed* sets, then the above two definitions are consistent with each other, that is, $\mathcal{R}_{\mathcal{H}}$ is exactly the set of states that are covered by $\mathcal{T}r(\mathcal{H})$.

One of the major concerned properties of hybrid systems is *safety.* Given an HA $\mathcal{H}$, a safety requirement $\mathcal{S}$ assigns to each mode $q \in Q$ a safe region $S_q \subseteq \mathbb{R}^n$, i.e. $\mathcal{S} = \bigcup_{q \in Q}(\{q\} \times S_q)$. We say that $\mathcal{H}$ satisfies $\mathcal{S}$ if $\mathbf{x} \in S_q$ for all $(q, \mathbf{x}) \in \mathcal{R}_{\mathcal{H}}$.

The prominent feature that distinguishes hybrid systems from traditional discrete programs and makes them more difficult to study is continuous behavior. To facilitate the investigation of continuous parts of hybrid systems, the following definition is proposed.

Definition 6 (Constrained CDS). *A constrained continuous dynamical system (CCDS) is a pair $(D, \mathbf{f})$, where $D \subseteq \mathbb{R}^n$ and $\mathbf{f}$ is a locally Lipschitz continuous vector field in $\mathbb{R}^n$.*

Thus an HA can be regarded as a composition of a finite set of CCDSs, one for each mode, together with discrete transitions among the CCDSs.

2.3 Polynomials and Polynomial Ideals

The tractability of the problems of analysis, verification and synthesis of hybrid systems depends on the language used to specify the hybrid systems, as well as the concerned properties. In this tutorial, we will focus on the class of *polynomial* expressions, which have powerful modeling ability and are easy to manipulate. We will give a brief overview of the theory of polynomials and polynomial ideals here. For more details please refer to [24].

A *monomial* in n variables $x_1, x_2, \dots, x_n$ (or briefly $\mathbf{x}$) is a product form $x_1^{\alpha_1} x_2^{\alpha_2} \cdots x_n^{\alpha_n}$, or briefly $\mathbf{x}^{\boldsymbol{\alpha}}$, where $\boldsymbol{\alpha} = (\alpha_1, \alpha_2, \dots, \alpha_n) \in \mathbb{N}^n$. The number $\sum_{i=1}^{n} \alpha_i$ is called the *degree* of $\mathbf{x}^{\boldsymbol{\alpha}}$.

Let $\mathbb{K}$ be a number field, which can be either $\mathbb{Q}$ or $\mathbb{R}$ in this tutorial. A *polynomial* $p(\mathbf{x})$ in $\mathbf{x}$ (or briefly p) with coefficients in $\mathbb{K}$ is of the form $\sum_{\boldsymbol{\alpha}} c_{\boldsymbol{\alpha}} \mathbf{x}^{\boldsymbol{\alpha}}$, where all $c_{\boldsymbol{\alpha}} \in \mathbb{K}$. The *degree* of p, denoted by $\mathsf{deg}(p)$, is the maximal degree of its component monomials. It is easy to see that a polynomial in $x_1, x_2, \dots, x_n$ with degree d has at most $\binom{n+d}{d}$ many coefficients. The set of all polynomials in $x_1, x_2, \dots, x_n$ with coefficients in $\mathbb{K}$ form a *polynomial ring*, denoted by $\mathbb{K}[\mathbf{x}]$.

A *parametric* polynomial is of the form $\sum_{\boldsymbol{\alpha}} u_{\boldsymbol{\alpha}} \mathbf{x}^{\boldsymbol{\alpha}}$, where $u_{\boldsymbol{\alpha}} \in \mathbb{R}$ are not constants but undetermined parameters. It can also be regarded as a standard polynomial $p(\mathbf{u}, \mathbf{x})$ in $\mathbb{Q}[\mathbf{u}, \mathbf{x}]$, where $\mathbf{u} = (u_1, u_2, \dots, u_w)$ is the set of all parameters. It is easy to see that a parametric polynomial with degree d (in $\mathbf{x}$) has at most $\binom{n+d}{d}$ many indeterminates. In practice, one only keeps some of the $u_{\boldsymbol{\alpha}}$'s as unknowns, by judiciously fixing the coefficients of specific monomials. For any $\mathbf{u}_0 \in \mathbb{R}^w$, we call $p_{\mathbf{u}_0}(\mathbf{x}) \in \mathbb{R}[\mathbf{x}]$, obtained by substituting $\mathbf{u}_0$ for $\mathbf{u}$ in $p(\mathbf{u}, \mathbf{x})$, an *instantiation* of $p(\mathbf{u}, \mathbf{x})$.

A vector field $\mathbf{f}$ is called a *polynomial vector field* (PVF) if each element of $\mathbf{f}$ is a polynomial. Given a polynomial $p \in \mathbb{K}[\mathbf{x}]$ and a PVS $\mathbf{f} \in \mathbb{K}^n[\mathbf{x}]$, according to Section 2.1, the Lie derivatives $L_{\mathbf{f}}^k p(\mathbf{x})$ is defined for all $k \in \mathbb{N}$ and all polynomials in $\mathbb{K}[\mathbf{x}]$. The Lie derivatives of a parametric polynomial $p(\mathbf{u}, \mathbf{x}) \in \mathbb{K}[\mathbf{u}, \mathbf{x}]$ can be defined similarly by setting the gradient as

$$\nabla p(\mathbf{u}, \mathbf{x}) \mathrel{\widehat{=}} \left(\frac{\partial p}{\partial x_1}, \frac{\partial p}{\partial x_2}, \cdots, \frac{\partial p}{\partial x_n}\right).$$

In this way all $L_{\mathbf{f}}^k p(\mathbf{u}, \mathbf{x})$ are still polynomials in $\mathbb{K}[\mathbf{u}, \mathbf{x}]$.

We next recall the basic theory of polynomial ideals.

Definition 7 (Polynomial Ideal). *A subset $I \subseteq \mathbb{K}[\mathbf{x}]$ is called an* **ideal** *if the following conditions are satisfied:*

1. *$0 \in I$;*
2. *If p, $g \in I$, then $p + g \in I$;*
3. *If $p \in I$ and $h \in \mathbb{K}[\mathbf{x}]$, then $hp \in I$.*

Let $g_1, g_2, \dots, g_s \in \mathbb{K}[\mathbf{x}]$. It is easy to check that the set

$$\langle g_1, g_2, \dots, g_s \rangle \mathrel{\widehat{=}} \left\{ \sum_{i=1}^{s} h_i g_i : h_1, h_2, \dots, h_s \in \mathbb{K}[\mathbf{x}] \right\}$$

is an ideal, called the ideal *generated* by $g_1, g_2, \dots, g_s$. If $I = \langle g_1, g_2, \dots, g_s \rangle$, then $\{g_1, g_2, \dots, g_s\}$ is called a *basis* of I.

Theorem 1 (Hilbert Basis Theorem). *Every ideal $I \subseteq \mathbb{K}[\mathbf{x}]$ has a basis, that is, $I = \langle g_1, g_2, \dots, g_s \rangle$ for some $g_1, g_2, \dots, g_s \in \mathbb{K}[\mathbf{x}]$.*

In particular, every ideal $I \subseteq \mathbb{K}[\mathbf{x}]$ has a *Gröbner basis* which possesses very nice properties. To illustrate this, we need to fix an ordering of monomials. First, suppose the list of variables $x_1, x_2, \ldots, x_n$ are ordered by $x_1 \succ x_2 \succ \cdots \succ x_n$. Then $\succ$ induces a total ordering on the set of monomials $\mathbf{x}^{\boldsymbol{\alpha}}$ with $\boldsymbol{\alpha} \in \mathbb{N}^n$. One example is the *lexicographic* (lex for short) order, i.e. $\mathbf{x}^{\boldsymbol{\alpha}} \succ \mathbf{x}^{\boldsymbol{\beta}}$ if and only if there exists $1 \leq i \leq n$ such that $\alpha_i > \beta_i$, and $\alpha_j = \beta_j$ for all $1 \leq j < i$. It can be shown that the lex order of monomials is a *well-ordering*, that is, every nonempty set of monomials has a *least* element. Besides, the lex order is preserved under multiplication, i.e. $\mathbf{x}^{\boldsymbol{\alpha}} \succ \mathbf{x}^{\boldsymbol{\beta}}$ implies $\mathbf{x}^{\boldsymbol{\alpha}}\mathbf{x}^{\boldsymbol{\gamma}} \succ \mathbf{x}^{\boldsymbol{\beta}}\mathbf{x}^{\boldsymbol{\gamma}}$ for any $\boldsymbol{\gamma} \in \mathbb{N}^n$. Such an ordering of monomials as the lex order is called a *monomial ordering.*

Given a monomial ordering $\succ$ and a polynomial $g \in \mathbb{K}[\mathbf{x}]$, rearrange the monomials in p in a descending order as

$$g = c_1\mathbf{x}^{\boldsymbol{\alpha_1}} + c_2\mathbf{x}^{\boldsymbol{\alpha_2}} + \cdots + c_k\mathbf{x}^{\boldsymbol{\alpha_k}} \enspace ,$$

where all c_i's are nonzero. Then $c_1\mathbf{x}^{\boldsymbol{\alpha}_1}$ is called the *leading term* of g, denoted by $\mathsf{lt}(g)$; c_1 is called the *leading coefficient* of g, denoted by $\mathsf{lc}(g)$; and $\mathbf{x}^{\boldsymbol{\alpha_1}}$ is called the *leading monomial* of g, denoted by $\mathsf{lm}(g)$. For a polynomial $p \in \mathbb{K}[\mathbf{x}]$, if p has a nonzero term $c_{\boldsymbol{\beta}}\mathbf{x}^{\boldsymbol{\beta}}$ and $\mathbf{x}^{\boldsymbol{\beta}}$ is divisible by $\mathsf{lm}(g)$, i.e. $\mathbf{x}^{\boldsymbol{\beta}} = \mathbf{x}^{\boldsymbol{\gamma}}\mathsf{lm}(g)$ for some $\boldsymbol{\gamma} \in \mathbb{N}^n$, then we say p is *reducible* modulo g, and call

$$p' = p - \frac{c_{\boldsymbol{\beta}}}{\mathsf{lc}(g)}\mathbf{x}^{\boldsymbol{\gamma}} g$$

the one-step *reduction* of p modulo g.

Given a finite set of polynomials $G \subsetneq \mathbb{K}[\mathbf{x}]$ and a polynomial $p \in \mathbb{K}[\mathbf{x}]$, we can do a muli-step reduction on p using polynomials in G, until p is reduced to p^* which is not further reducible modulo G. Such p^* is called the *normal form* of p w.r.t. G, denoted by $\mathsf{nf}(p, G)$. For general G, the above process of reduction is guaranteed to terminate; however, the final result $\mathsf{nf}(p, G)$ may vary, depending on the sequence of polynomials chosen from G during reduction. Fortunately, we have

Proposition 1. *Given a monomial ordering, then every ideal $I \subseteq \mathbb{K}[\mathbf{x}]$ other than $\{0\}$ has a basis $G = \{g_1, g_2, \ldots, g_s\}$, such that for any $p \in \mathbb{K}[\mathbf{x}]$, $\mathsf{nf}(p, G)$ is unique. Such G is called a* **Gröbner basis** *of I.*

Furthermore,

Proposition 2. *Let G be a Gröbner basis of an ideal $I \subseteq \mathbb{K}[\mathbf{x}]$. Then for any $p \in \mathbb{K}[\mathbf{x}]$, $p \in I$ if and only if $\mathsf{nf}(p, G) = 0$.*

Most importantly, for any ideal $I = \langle h_1, h_2, \ldots, h_l\rangle \subseteq \mathbb{K}[\mathbf{x}]$, the Gröbner basis G of I can be computed from the h_i's using *Buchberger's Algorithm* [24]. Then by Proposition 2, we get that the *ideal membership* problem, that is to decide whether a polynomial $p \in \mathbb{K}[\mathbf{x}]$ lies in a given ideal $\langle h_1, h_2, \ldots, h_l\rangle \subseteq \mathbb{K}[\mathbf{x}]$, is algorithmically solvable.

The following theorem, which can be deduced from Hilbert Basis Theorem, is key to the proof of several main results in this tutorial.

Theorem 2 (Ascending Chain Condition). *For any ascending chain of ideals*

$$I_1 \subseteq I_2 \subseteq \cdots \subseteq I_l \subseteq \cdots$$

in $\mathbb{K}[\mathbf{x}]$*, there exists an* $N \in \mathbb{N}$ *such that* $I_l = I_N$ *for any* $l \geq N$*.*

2.4 First-Order Theory of Reals

From a logical point of view, polynomials can be used to construct the first-order theory $T(\mathbb{R})$ of *real numbers* (actually of all *real closed fields*), which is very useful in formulating problems arising in the study of hybrid systems. The language of $T(\mathbb{R})$ consists of

- variables: $x, y, z, \ldots, x_1, x_2, \ldots$, which are interpreted over $\mathbb{R}$;
- relational symbols: $>, <, \geq, \leq, =, \neq$;
- Boolean connectives: $\wedge, \vee, \neg, \rightarrow, \leftrightarrow, \ldots$; and
- quantifiers: $\forall, \exists$.

A *term* of $T(\mathbb{R})$ over a finite set of variables $\{x_1, x_2, \ldots, x_n\}$ is a polynomial $p \in \mathbb{Q}[x_1, x_2, \ldots, x_n]$. An *atomic formula* of $T(\mathbb{R})$ is of the form $p \rhd 0$, where $\rhd$ is any relational symbol. A *quantifier-free formula* (QFF) of $T(\mathbb{R})$ is a Boolean combination of atomic formulas. A generic formula of $T(\mathbb{R})$ is built up from atomic formulas using Boolean connectives as well as quantifiers.

A profound result about $T(\mathbb{R})$ is that it admits *quantifier elimination* (QE) [83]. That is, any formula φ in $T(\mathbb{R})$ has a quantifier-free equivalent φ_{QF} involving only *free* variables of φ, and φ_{QF} can be computed from φ using QE algorithms. An immediate consequence of this result is the *decidability* of $T(\mathbb{R})$: the truth value of any formula in $T(\mathbb{R})$ can be decided.

Formulas in $T(\mathbb{R})$ define a special class of sets:

Definition 8 (Semi-algebraic Set). *A subset* $A \subseteq \mathbb{R}^n$ *is called a* semi-algebraic set *(SAS), if there exists a QFF* ϕ *in* $T(\mathbb{R})$ *over variables* $x_1, x_2, \ldots, x_n$*, or briefly* $\mathbf{x}$*, such that*

$$A = \{\mathbf{x} \in \mathbb{R}^n \mid \phi(\mathbf{x}) \textit{ is true}\} \ .$$

Let $\mathcal{A}(\phi)$ denote the SAS defined by a QFF ϕ. Then from Definition 8 it is easy to check that SASs are closed under common set operations:

- $\mathcal{A}(\phi_1) \cap \mathcal{A}(\phi_2) = \mathcal{A}(\varphi_1 \wedge \varphi_2)$;
- $\mathcal{A}(\phi_1) \cup \mathcal{A}(\phi_2) = \mathcal{A}(\varphi_1 \vee \phi_2)$;
- $\mathcal{A}(\phi_1)^c = \mathcal{A}(\neg\phi_1)$;
- $\mathcal{A}(\phi_1) \setminus \mathcal{A}(\phi_2) = \mathcal{A}(\phi_1) \cap \mathcal{A}(\phi_2)^c = \mathcal{A}(\phi_1 \wedge \neg\phi_2)$,

where A^c and $A \setminus B$ stand for the complement and subtraction operation of sets respectively. Moreover, checking of emptiness, inclusion and equality of SASs can be done by the decidability of $T(\mathbb{R})$.

For convenience, in the rest of this tutorial, we do not distinguish between an SAS $\mathcal{A}(\phi)$ and its defining formula ϕ. That is, we will use $T(\mathbb{R})$-formulas to represent SASs and use Boolean connectives as set operators. Besides, it is easy to check that any SAS can be represented by a QFF in the form of

$$\phi(\mathbf{x}) \widehat{=} \bigvee_{k=1}^{K} \bigwedge_{j=1}^{J_k} p_{kj}(\mathbf{x}) \triangleright 0\,,$$

where $p_{kj}(\mathbf{x}) \in \mathbb{Q}[\mathbf{x}]$ and $\triangleright \in \{\geq, >\}$. Therefore restricting SASs to formulas of this shape will not lose any generality.

Definition 9 (Semi-algebraic Template). *A* semi-algebraic template *with degree d is of the form*

$$\phi(\mathbf{u}, \mathbf{x}) \widehat{=} \bigvee_{k=1}^{K} \bigwedge_{j=1}^{J_k} p_{kj}(\mathbf{u}_{kj}, \mathbf{x}) \triangleright 0\,,$$

where $p_{kj} \in \mathbb{Q}[\mathbf{u}_{kj}, \mathbf{x}]$ are parametric polynomials with degree d (in $\mathbf{x}$), $\mathbf{u}$ is the collection of parameters appearing in each p_{kj} (i.e. $\mathbf{u}_{kj}$), and $\triangleright \in \{\geq, >\}$.

As mentioned in Section 2.3, we will focus on hybrid systems and properties described by polynomial expressions.

Definition 10. *A polynomial CDS (or CCDS, HA, safety property, etc), denoted by PCDS (or PCCDS, PHA, etc) for short, is a CDS (or CCDS, HA, safety property etc, respectively) wherein the sets are SASs and the vector fields are PVFs (with rational coefficients).*

3 Computing Invariants for Hybrid Systems

3.1 Continuous and Global Invariant

An *invariant* of a hybrid system is a property that holds at every reachable state of the system.

Definition 11 (Invariant). *An invariant of an HA $\mathcal{H}$ maps to each $q \in Q$ a subset $I_q \subseteq \mathbb{R}^n$, such that for all $(q, \mathbf{x}) \in \mathcal{R}_{\mathcal{H}}$, we have $\mathbf{x} \in I_q$.*

One effective way of finding invariants of hybrid systems is to generate so-called *inductive invariants*, as inductiveness is usually checkable [75].

Definition 12 (Inductive Invariant). *Given an HA $\mathcal{H}$, an inductive invariant maps to each $q \in Q$ a subset $I_q \subseteq \mathbb{R}^n$, such that the following conditions are satisfied:*

1. *$\Xi_q \subseteq I_q$ for all $q \in Q$;*
2. *for any $e = (q, q') \in E$, if $\mathbf{x} \in I_q \cap G_e$, then $\mathbf{x}' = R_e(\mathbf{x}) \in I_{q'}$;*
3. *for any $q \in Q$ and any $\mathbf{x}_0 \in I_q$, if there exists a $\delta \geq 0$ s.t. the solution $\mathbf{x}(\mathbf{x}_0; t)$ to $\dot{\mathbf{x}} = \mathbf{f}_q$ satisfies: (i) $\mathbf{x}(\mathbf{x}_0; \delta) = \mathbf{x}'$; and (ii) $\mathbf{x}(\mathbf{x}_0; t) \in D_q$ for all $t \in [0, \delta]$, then $\mathbf{x}' \in I_q$.*

It is easy to check that any inductive invariant is also an invariant. We assume in this section that all invariants mentioned are *inductive*.

In Definition 12, condition 1 and 2 are about initial states and discrete inductiveness, which can be checked using the standard techniques for the verification of discrete programs [92]. However, it is not so straightforward and requires special efforts to check condition 3, for which the notion of *continuous invariant*[2] [65,56] is quite useful.

Definition 13 (Continuous Invariant). *A subset $I \subseteq \mathbb{R}^n$ is called a* continuous invariant *(CI) of a CCDS $(D, \mathbf{f})$ if for any $\mathbf{x}_0 \in I$ and any $T \geq 0$, we have:*

$$(\forall t \in [0,T].\, \mathbf{x}(\mathbf{x}_0;t) \in D) \Longrightarrow (\forall t \in [0,T].\, \mathbf{x}(\mathbf{x}_0;t) \in I)\,,$$

or equivalently,

$$(\forall t \in [0,T].\, \mathbf{x}(\mathbf{x}_0;t) \in D) \Longrightarrow \mathbf{x}(\mathbf{x}_0;T) \in I\,.$$

By Definition 13, it is not difficult to check that condition 3 in Definition 12 is equivalent to

3'. for any $q \in Q$, I_q is a CI of $(D_q, \mathbf{f}_q)$.

To distinguish from CI, we refer to the inductive invariant in Definition 12 a *global invariant* (GI). Simply, a GI of an HA $\mathcal{H}$ consists of a set of CIs, one for each CCDS corresponding to a mode of the HA. Using GI, if $I_q \subseteq S_q$ for all q, then a safety property $\mathcal{S}$ can be verified without computing $\mathcal{R}_{\mathcal{H}}$. In the rest of this section, we will present an approach for automatically discovering semi-algebraic CIs (SCI) and semi-algebraic GIs (SGI) for PCCDS and PHA respectively.

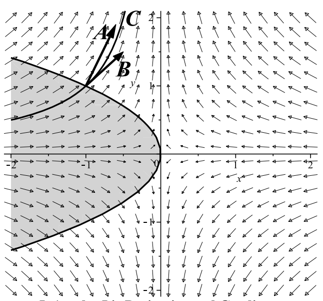
3.2 Predicting Continuous Evolution via Lie Derivatives

Given a PVF $\mathbf{f}$, we can make use of Lie derivatives to investigate the tendency of $\mathbf{f}$'s trajectories in terms of a polynomial p. To capture this, look at Example 1 shown in I of Figure 1.

Example 1. Suppose $\mathbf{f} = (-x, y)$ and $p(x,y) = x + y^2$. Then

$$\begin{aligned} L_{\mathbf{f}}^0 p(x,y) &= x + y^2 \\ L_{\mathbf{f}}^1 p(x,y) &= -x + 2y^2 \\ L_{\mathbf{f}}^2 p(x,y) &= x + 4y^2 \\ &\vdots \end{aligned}$$

[2] In some later sections of this tutorial when we talk about the Hybrid Hoare Logic (HHL), the terminology *differential invariant* is used instead of continuous invariant, with exactly the same meaning.



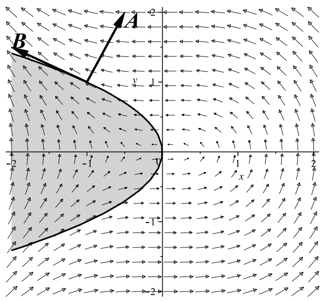


Fig. 1. Lie Derivatives

In I of Figure 1, vector B denotes the corresponding evolution direction of the vector field $\mathbf{f} = (-x, y)$ at point $(-1, 1)$. We could imagine the points on the parabola $p(x, y) = x + y^2$ with zero energy, and the points in the white area have positive energy, i.e. $p(x, y) > 0$. Vector A is the gradient $\nabla p|_{(-1,1)}$ of $p(x, y)$, which infers that the trajectory starting at $(-1, 1)$ will enter the white area immediately if the angle, between $\nabla p|_{(-1,1)}$ and the evolution direction at $(-1, 1)$, is less than $\frac{\pi}{2}$, which means equivalently that the 1-order Lie derivative $L^1_{\mathbf{f}} p|_{(-1,1)}$ is positive. Thus the 1-order Lie derivative $L^1_{\mathbf{f}} p|_{(-1,1)} = 3$ predicts that there is some positive $\epsilon > 0$ such that the trajectory starting at $(-1, 1)$ (curve C) has the property $p\Big(\mathbf{x}((-1, 1); t)\Big) > 0$ for all $t \in (0, \epsilon)$.

However, if the angle between the gradient and the evolution direction equals $\frac{\pi}{2}$ or the gradient is zero-vector, then the 1-order Lie derivative is zero and it is impossible to predict trajectory tendency by means of 1-order Lie derivative. In this case, we resort to nonzero higher order Lie derivatives. For this purpose, we introduce the *pointwise rank* of p with respect to $\mathbf{f}$ as the function $\gamma_{p,\mathbf{f}} : \mathbb{R}^n \to \mathbb{N} \cup \{\infty\}$ defined by

$$\gamma_{p,\mathbf{f}}(\mathbf{x}) = \min\{k \in \mathbb{N} \mid L^k_{\mathbf{f}} p(\mathbf{x}) \neq 0\}$$

if such k exists, and $\gamma_{p,\mathbf{f}}(\mathbf{x}) = \infty$ otherwise.

Example 2. Let $\mathbf{f}(x, y) = (-2y, x^2)$ and $h(x, y) = x + y^2$. Then

$$\begin{aligned} L^0_{\mathbf{f}} h(x, y) &= x + y^2 \\ L^1_{\mathbf{f}} h(x, y) &= -2y + 2x^2 y \\ L^2_{\mathbf{f}} h(x, y) &= -8y^2 x - (2 - 2x^2)x^2 \\ &\vdots \end{aligned}$$

Here, $\gamma_{h,\mathbf{f}}(0, 0) = \infty$, $\gamma_{h,\mathbf{f}}(-4, 2) = 1$, etc.

Look at II of Figure 1. At point $(-1,1)$ on curve $h(x,y)=0$, the gradient of h is $(1,2)$ (vector A) and the evolution direction is $(-2,1)$ (vector B), so their inner product is zero. Thus it is impossible to predict the tendency (in terms of curve $h(x,y)=0$) of the trajectory starting from $(-1,1)$ via the 1-order Lie derivative. By a simple computation, the 2-order Lie derivative $L_{\mathbf{f}}^2 h(-1,1)$ is 8. Hence $\gamma_{h,\mathbf{f}}(-1,1)=2$. In the sequel, we shall show how to use such high order Lie derivatives to analyze the trajectory tendency.

For analyzing trajectory tendency by high order Lie derivatives, we need the following fact.

Proposition 3. *Given a PVF* $\mathbf{f}$ *and a polynomial* p*, then for any* $\mathbf{x}_0 \in \mathbb{R}^n$*,* $p(\mathbf{x}_0)=0$ *if and only if* $\gamma_{p,\mathbf{f}}(\mathbf{x}_0) \neq 0$*. Let* $\mathbf{x}(t) \,\widehat{=}\, \mathbf{x}(\mathbf{x}_0;t)$*. Then it follows that*

(a) if $\gamma_{p,\mathbf{f}}(\mathbf{x}_0) < \infty$ *and* $L_{\mathbf{f}}^{\gamma_{p,\mathbf{f}}(\mathbf{x}_0)} p(\mathbf{x}_0) > 0$*, then*

$$\exists \epsilon > 0, \forall t \in (0,\epsilon).\, p(\mathbf{x}(t)) > 0;$$

(b) if $\gamma_{p,\mathbf{f}}(\mathbf{x}_0) < \infty$ *and* $L_{\mathbf{f}}^{\gamma_{p,\mathbf{f}}(\mathbf{x}_0)} p(\mathbf{x}_0) < 0$*, then*

$$\exists \epsilon > 0, \forall t \in (0,\epsilon).\, p(\mathbf{x}(t)) < 0;$$

(c) if $\gamma_{p,\mathbf{f}}(\mathbf{x}_0) = \infty$*, then*

$$\exists \epsilon > 0, \forall t \in (0,\epsilon).\, p(\mathbf{x}(t)) = 0.$$

Proof. By Section 2.1, $p(\mathbf{x}(t))$ is the composition of two analytic functions, which implies [52] that the Taylor expansion of $p(\mathbf{x}(t))$ at $t=0$

$$\begin{aligned} p(\mathbf{x}(t)) &= p(\mathbf{x}_0) + \frac{\mathrm{d}p}{\mathrm{d}t}\cdot t + \frac{\mathrm{d}^2 p}{\mathrm{d}t^2}\cdot\frac{t^2}{2!} + \cdots \\ &= L_{\mathbf{f}}^0 p(\mathbf{x}_0) + L_{\mathbf{f}}^1 p(\mathbf{x}_0)\cdot t + L_{\mathbf{f}}^2 p(\mathbf{x}_0)\cdot\frac{t^2}{2!} + \cdots \end{aligned} \quad (2)$$

converges in a neighborhood of zero. Then the conclusion of Proposition 3 follows immediately from formula (2) by case analysis on the sign of $L_{\mathbf{f}}^{\gamma_{p,\mathbf{f}}(\mathbf{x}_0)} p(\mathbf{x}_0)$. □

Based on this proposition, we introduce the notion of *transverse set* to indicate the tendency of the trajectories of a considered PVF in terms of the first nonzero high order Lie derivative of an underlying polynomial as follows.

Definition 14 (Transverse Set). *Given a polynomial* p *and a PVF* $\mathbf{f}$*, the transverse set of* $\mathbf{f}$ *over the domain* $P \,\widehat{=}\, p(\mathbf{x}) \geq 0$ *is*

$$Trans_{\mathbf{f}\uparrow p} \,\widehat{=}\, \{\mathbf{x} \in \mathbb{R}^n \mid \gamma_{p,\mathbf{f}}(\mathbf{x}) < \infty \wedge L_{\mathbf{f}}^{\gamma_{p,\mathbf{f}}(\mathbf{x})} p(\mathbf{x}) < 0\}.$$

Intuitively, if $\mathbf{x} \in Trans_{\mathbf{f}\uparrow p}$, then either $\mathbf{x}$ is not in P, or $\mathbf{x}$ is on the boundary of P (i.e. $p(\mathbf{x})=0$) such that the trajectory $\mathbf{x}(t)$ starting from $\mathbf{x}$ will exit P immediately.

3.3 Computing Transverse Set

The set $Trans_{\mathbf{f}\uparrow p}$ in Definition 14 plays a crucial role in developing the automatic invariant generation method. First of all, we have

Theorem 3. *Given a polynomial $p \in \mathbb{Q}[\mathbf{x}]$ and a PVF $\mathbf{f} \in \mathbb{Q}^n[\mathbf{x}]$, the set $Trans_{\mathbf{f}\uparrow p}$ is an SAS, and its explicit representation is computable.*

To prove this theorem, it suffices to show $\gamma_{p,\mathbf{f}}(\mathbf{x})$ is computable for each $\mathbf{x} \in \mathbb{R}^n$. However, $\gamma_{p,\mathbf{f}}(\mathbf{x})$ may be infinite for some $\mathbf{x}$. Thus, it seems that we have to compute $L_{\mathbf{f}}^k p(\mathbf{x})$ infinitely many times for such $\mathbf{x}$ to determine if $\mathbf{x} \in Trans_{\mathbf{f}\uparrow p}$. Fortunately, we can find a uniform upper bound on $\gamma_{p,\mathbf{f}}(\mathbf{x})$ for all $\mathbf{x}$ with finite pointwise rank. To see this, consider the polynomial ideals in ring $\mathbb{Q}[\mathbf{x}]$ generated by Lie derivatives $L_{\mathbf{f}}^0 p, L_{\mathbf{f}}^1 p, \ldots, L_{\mathbf{f}}^i p$ for all $i \geq 0$, i.e.

$$J_i \mathrel{\widehat{=}} \langle L_{\mathbf{f}}^0 p(\mathbf{x}), L_{\mathbf{f}}^1 p(\mathbf{x}), \ldots, L_{\mathbf{f}}^i p(\mathbf{x}) \rangle \,.$$

Note that

$$J_0 \subseteq J_1 \subseteq \cdots \subseteq J_l \subseteq \cdots$$

forms an ascending chain of ideals in $\mathbb{Q}[\mathbf{x}]$. By Theorem 2, the number

$$N_{p,\mathbf{f}} \mathrel{\widehat{=}} \min\{i \in \mathbb{N} \mid J_i = J_{i+1}\}, \tag{3}$$

or equivalently,

$$N_{p,\mathbf{f}} \mathrel{\widehat{=}} \min\{i \in \mathbb{N} \mid L_{\mathbf{f}}^{i+1} p \in J_i\}$$

is well-defined. Furthermore, $N_{p,\mathbf{f}}$ can be computed by solving the ideal membership problem with the assistance of an algebraic tool like Maple [61].

Example 3. For $\mathbf{f}$ and h in Example 2, by simple computations we get $L_{\mathbf{f}}^1 h \notin \langle L_{\mathbf{f}}^0 h \rangle$, $L_{\mathbf{f}}^2 h \notin \langle L_{\mathbf{f}}^0 h, L_{\mathbf{f}}^1 h \rangle$, $L_{\mathbf{f}}^3 h \in \langle L_{\mathbf{f}}^0 h, L_{\mathbf{f}}^1 h, L_{\mathbf{f}}^2 h \rangle$, so $N_{h,\mathbf{f}} = 2$.

Actually, the integer $N_{p,\mathbf{f}}$ is the upper bound mentioned above on pointwise rank by the following two theorems.

Theorem 4 (Fixed Point Theorem). *If $J_i = J_{i+1}$, then $J_i = J_l$ for all $l > i$.*

Proof. We prove this fact by induction on l. Base case: $J_i = J_{i+1}$. Assume $J_i = J_l$ for some $l \geq i+1$. Then there are $g_j \in \mathbb{Q}[\mathbf{x}]$ for $0 \leq j \leq i$, such that $L_{\mathbf{f}}^l p = \sum_{j=0}^{i} g_j L_{\mathbf{f}}^j p$. By the definition of Lie derivatives it follows that

$$\begin{aligned}
L_{\mathbf{f}}^{l+1}p &= (\nabla L_{\mathbf{f}}^{l}p, \mathbf{f}) \\
&= (\nabla \sum_{j=0}^{i} g_j L_{\mathbf{f}}^{j}p, \mathbf{f}) \\
&= (\sum_{j=0}^{i} L_{\mathbf{f}}^{j}p \nabla g_j + \sum_{j=0}^{i} g_j \nabla L_{\mathbf{f}}^{j}p, \mathbf{f}) \\
&= \sum_{j=0}^{i} (\nabla g_j, \mathbf{f}) L_{\mathbf{f}}^{j}p + \sum_{j=0}^{i} g_j L_{\mathbf{f}}^{j+1}p \\
&= \sum_{j=0}^{i} (\nabla g_j, \mathbf{f}) L_{\mathbf{f}}^{j}p + \sum_{j=1}^{i} g_{j-1} L_{\mathbf{f}}^{j}p + g_i L_{\mathbf{f}}^{i+1}p \,. \qquad (4)
\end{aligned}$$

By base case, $L_{\mathbf{f}}^{i+1}p \in J_i$. Then by (4) we get $L_{\mathbf{f}}^{l+1}p \in J_i$, so $J_i = J_{l+1}$. By induction, the fact follows immediately. □

Theorem 5 (Rank Theorem). *Given a polynomial p and a PVF $\mathbf{f}$, for any $\mathbf{x} \in \mathbb{R}^n$, if $\gamma_{p,\mathbf{f}}(\mathbf{x}) < \infty$, then $\gamma_{p,\mathbf{f}}(\mathbf{x}) \le N_{p,\mathbf{f}}$, where $N_{p,\mathbf{f}}$ is defined in (3).*

Proof. If $N_{p,\mathbf{f}} < \gamma_{p,\mathbf{f}}(\mathbf{x}) < \infty$, then $\bigwedge_{i=0}^{N_{p,\mathbf{f}}} L_{\mathbf{f}}^{i}p(\mathbf{x}) = 0$. By (3) and Theorem 4 we get $L_{\mathbf{f}}^{i}p(\mathbf{x}) = 0$ for all $i \in \mathbb{N}$. Thus $\gamma_{p,\mathbf{f}}(\mathbf{x}) = \infty$, which is a contradiction. □

Now, applying the above two theorems we can prove Theorem 3.

Proof (of Theorem 3). First by Theorem 5, for any $\mathbf{x}$,

$$\mathbf{x} \in Trans_{\mathbf{f}\uparrow p} \Longleftrightarrow \gamma_{p,\mathbf{f}}(\mathbf{x}) \le N_{p,\mathbf{f}} \wedge L_{\mathbf{f}}^{\gamma_{p,\mathbf{f}}(\mathbf{x})}p(\mathbf{x}) < 0 \,. \qquad (5)$$

Given p and $\mathbf{f}$, let

$$\pi^{(0)}(p, \mathbf{f}, \mathbf{x}) \widehat{=} \; p(\mathbf{x}) < 0 \,;$$

for $1 \le i \in \mathbb{N}$,

$$\pi^{(i)}(p, \mathbf{f}, \mathbf{x}) \,\widehat{=}\, \left(\bigwedge_{0 \le j < i} L_{\mathbf{f}}^{j}p(\mathbf{x}) = 0 \right) \wedge L_{\mathbf{f}}^{i}p(\mathbf{x}) < 0 \,,$$

and

$$\pi(p, \mathbf{f}, \mathbf{x}) \,\widehat{=} \bigvee_{0 \le i \le N_{p,\mathbf{f}}} \pi^{(i)}(p, \mathbf{f}, \mathbf{x}) \,.$$

Then from (5) we have another equivalence

$$\mathbf{x} \in Trans_{\mathbf{f}\uparrow p} \Longleftrightarrow \pi(p, \mathbf{f}, \mathbf{x}) \,. \qquad (6)$$

Thus $Trans_{\mathbf{f}\uparrow p}$ is actually an SAS which can be represented by $\pi(p, \mathbf{f}, \mathbf{x})$. □

In automatic invariant generation, it actually makes use of parametric polynomials $p(\mathbf{u},\mathbf{x})$. The following theorem indicates Theorem 5 still holds after substituting $p(\mathbf{u},\mathbf{x})$ for $p(\mathbf{x})$.

Theorem 6 (Parametric Rank Theorem). *Given a parametric polynomial $p(\mathbf{u},\mathbf{x})$ and a PVF $\mathbf{f}$, there is an integer $N_{p,\mathbf{f}} \in \mathbb{N}$ such that $\gamma_{p_{\mathbf{u}_0},\mathbf{f}}(\mathbf{x}) < \infty$ implies $\gamma_{p_{\mathbf{u}_0},\mathbf{f}}(\mathbf{x}) \le N_{p,\mathbf{f}}$ for all $\mathbf{x} \in \mathbb{R}^n$ and all $\mathbf{u}_0 \in \mathbb{R}^w$.*

The proof of this theorem is quite close to the one of Theorem 5. The difference lies in the settings of polynomials. Here, all polynomials and ideals are considered in the polynomial ring $\mathbb{Q}[\mathbf{u},\mathbf{x}]$, and the number $N_{p,\mathbf{f}}$ is defined similarly as in (3).

3.4 Computing SCI in Simple Case

Given a PCCDS $(D,\mathbf{f})$, the task is to find SCIs for $(D,\mathbf{f})$. First of all, we illustrate how to compute an SCI of the simple form $P \widehat{=} p(\mathbf{x}) \ge 0$ for a simple domain $D \widehat{=} h(\mathbf{x}) \ge 0$.

Notice that if $\mathbf{x}_0$ is in the interior of $P \cap D$, then the trajectory $\mathbf{x}(t)$ starting at $\mathbf{x}_0$ will remain in the interior within adequately small $t > 0$. Therefore, the condition of CI could be violated only at the points $\mathbf{x}$ on the boundary of P, i.e. $p(\mathbf{x}) = 0$. Thus by Definition 14 and Proposition 3, P is an invariant of $(D,\mathbf{f})$ if and only if for all $\mathbf{x}$

$$p(\mathbf{x}) = 0 \rightarrow \mathbf{x} \notin (Trans_{\mathbf{f}\uparrow p} \setminus Trans_{\mathbf{f}\uparrow h}),$$

i.e.

$$p(\mathbf{x}) = 0 \rightarrow \mathbf{x} \in (Trans_{\mathbf{f}\uparrow p})^c \cup Trans_{\mathbf{f}\uparrow h}. \tag{7}$$

By equivalence (6), the formula (7) is equivalent to

$$p(\mathbf{x}) = 0 \rightarrow (\neg\pi(p,\mathbf{f},\mathbf{x}) \vee \pi(h,\mathbf{f},\mathbf{x})),$$

i.e.

$$\big(p(\mathbf{x}) = 0 \wedge \pi(p,\mathbf{f},\mathbf{x})\big) \rightarrow \pi(h,\mathbf{f},\mathbf{x}). \tag{8}$$

Let $\theta(h,p,\mathbf{f},\mathbf{x})$ denote the formula (8). Then we obtain the following sufficient and necessary condition for P being an SCI of $(D,\mathbf{f})$.

Theorem 7 (Criterion Theorem). *Given a polynomial p, $p(\mathbf{x}) \ge 0$ is an SCI of the PCCDS $(h(\mathbf{x}) \ge 0, \mathbf{f})$ if and only if the formula $\theta(h,p,\mathbf{f},\mathbf{x})$ defined as (8) is true for all $\mathbf{x} \in \mathbb{R}^n$.*

Based on Theorem 7, a constraint based method for generating SCIs in the simple form can be presented as follows.

I. First, set a simple semi-algebraic template $P \widehat{=} p(\mathbf{u}, \mathbf{x}) \geq 0$ using a parametric polynomial $p(\mathbf{u}, \mathbf{x})$.

II. Then apply QE[3] to the formula $\forall \mathbf{x}.\theta(h, p, \mathbf{f}, \mathbf{x})$. In practice, QE may be applied to a formula $\forall \mathbf{x}.(\theta \wedge \phi)$, where ϕ is a formula imposing some additional constraint on the SCI P. If the output of QE is *false*, then there is no SCI in the form of the predefined P; otherwise, a constraint on $\mathbf{u}$, denoted by $R(\mathbf{u})$, will be returned.

III. Now, use an SMT solver like [26] to pick a $\mathbf{u}_0 \in R(\mathbf{u})$ and then $p_{\mathbf{u}_0}(\mathbf{x}) \geq 0$ is an SCI of $(h(\mathbf{x}) \geq 0, \mathbf{f})$ by Theorem 7.

Example 4. Again, we make use of Example 2 to demonstrate the above method. Here, we take $D \widehat{=} h(x, y) \geq 0$ with $h(x, y) \widehat{=} -x - y^2$ as the domain.

Apply procedure (I-III), we have:

1. Set a template $P \widehat{=} p(\mathbf{u}, \mathbf{x}) \geq 0$ with $p(\mathbf{u}, \mathbf{x}) \widehat{=} ay(x - y)$, where $\mathbf{u} \widehat{=} (a)$. By a simple computation we get $N_{p,\mathbf{f}} = 2$.
2. Compute the corresponding formula

$$\theta(h, p, \mathbf{f}, \mathbf{x}) \widehat{=} p = 0 \wedge (\pi^{(0)}_{p,\mathbf{f},\mathbf{x}} \vee \pi^{(1)}_{p,\mathbf{f},\mathbf{x}} \vee \pi^{(2)}_{p,\mathbf{f},\mathbf{x}}) \longrightarrow (\pi^{(0)}_{h,\mathbf{f},\mathbf{x}} \vee \pi^{(1)}_{h,\mathbf{f},\mathbf{x}} \vee \pi^{(2)}_{h,\mathbf{f},\mathbf{x}})$$

where

$$\begin{aligned}
\pi^{(0)}_{h,\mathbf{f},\mathbf{x}} &\widehat{=} -x - y^2 < 0, \\
\pi^{(1)}_{h,\mathbf{f},\mathbf{x}} &\widehat{=} -x - y^2 = 0 \wedge 2y - 2x^2y < 0, \\
\pi^{(2)}_{h,\mathbf{f},\mathbf{x}} &\widehat{=} -x - y^2 = 0 \wedge 2y - 2x^2y = 0 \ \wedge 8xy^2 + 2x^2 - 2x^4 < 0, \\
\pi^{(0)}_{p,\mathbf{f},\mathbf{x}} &\widehat{=} ay(x - y) < 0, \\
\pi^{(1)}_{p,\mathbf{f},\mathbf{x}} &\widehat{=} ay(x - y) = 0 \wedge -2ay^2 + ax^3 - 2yax^2 < 0, \\
\pi^{(2)}_{p,\mathbf{f},\mathbf{x}} &\widehat{=} ay(x - y) = 0 \wedge -2ay^2 + ax^3 - 2yax^2 = 0 \\
&\quad \wedge 40axy^2 - 16ay^3 + 32ax^3y - 10ax^4 < 0.
\end{aligned}$$

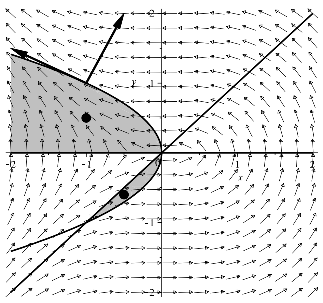
In addition, we require the two points $\{(-1, 0.5), (-0.5, -0.6)\}$ to be contained in P. Then apply QE to the formula

$$\forall x \forall y. \Big(\theta(h, p, \mathbf{f}, \mathbf{x}) \wedge 0.5a(-1 - 0.5) \geq 0 \ \wedge -0.6a(-0.5 + 0.6) \geq 0\Big) \ .$$

The result is $a \leq 0$.

3. Just pick $a = -1$, and then $-xy + y^2 \geq 0$ is an SCI of $(D, \mathbf{f})$. The grey part of Picture III in Fig. 2 is the intersection of the invariant P and domain D.

[3] QE has been implemented in many computer algebra tools such as QEPCAD [17], Redlog [30], Mathematica [88], etc.


III: SCI in Simple Case

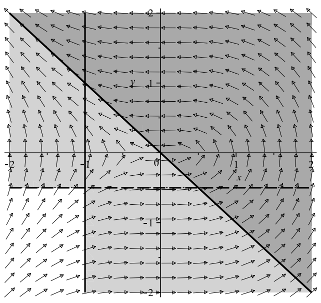

IV: SCI in General Case

Fig. 2. Semi-Algebraic Continuous Invariants

3.5 Computing SCI in General Case

Now, consider how to automatically discover SCIs of a PCCDS in general case. Given a PCCDS $(D, \mathbf{f})$ with

$$D \mathrel{\widehat{=}} \bigvee_{m=1}^{M} \bigwedge_{l=1}^{L_m} p_{ml}(\mathbf{x}) \rhd 0 \quad \text{with} \quad \rhd \in \{\geq, >\} \ , \tag{9}$$

the procedure of automatically generating SCIs with a general template

$$P \mathrel{\widehat{=}} \bigvee_{k=1}^{K} \bigwedge_{j=1}^{J_k} p_{kj}(\mathbf{u}_{kj}, \mathbf{x}) \rhd 0 \quad \text{with} \quad \rhd \in \{\geq, >\}$$

for $(D, \mathbf{f})$, is essentially the same as the steps (I-III) depicted in Section 3.4. However, we must sophisticatedly handle the complex Boolean structures of the formulas herein. In what follows, the main results on general SCI generation are outlined without rigorous proofs. Please refer to [54] for details.

Necessary-Sufficient Condition for CI. First of all, we study a necessary and sufficient condition like formula (7) for P being a CI of $(D, \mathbf{f})$. To analyze the evolution tendency of trajectories of $\mathbf{f}$ in terms of a subset $A \subseteq \mathbb{R}^n$, the following notions and notations are needed.

$$\mathrm{In}_{\mathbf{f}}(A) \mathrel{\widehat{=}} \{\mathbf{x}_0 \in \mathbb{R}^n \mid \exists \epsilon > 0 \forall t \in (0, \epsilon).\, \mathbf{x}(\mathbf{x}_0; t) \in A\},$$
$$\mathrm{IvIn}_{\mathbf{f}}(A) \mathrel{\widehat{=}} \{\mathbf{x}_0 \in \mathbb{R}^n \mid \exists \epsilon > 0 \forall t \in (0, \epsilon).\, \mathbf{x}(\mathbf{x}_0; -t) \in A\}.$$

Intuitively, $\mathbf{x}_0 \in \mathrm{In}_{\mathbf{f}}(A)$ means that the trajectory starting from $\mathbf{x}_0$ enters A immediately and keeps inside A for a certain amount of time; $\mathbf{x}_0 \in \mathrm{IvIn}_{\mathbf{f}}(A)$ means that the trajectory through $\mathbf{x}_0$ reaches $\mathbf{x}_0$ from inside A. By the notion of CI, it can be proved that

Theorem 8. *Given a CCDS $(D, \mathbf{f})$, a subset $P \subseteq \mathbb{R}^n$ is a CI of $(D, \mathbf{f})$ if and only if*

1. $\forall \mathbf{x} \in P \cap D \cap \mathrm{In}_{\mathbf{f}}(D).\, \mathbf{x} \in \mathrm{In}_{\mathbf{f}}(P)$; *and*
2. $\forall \mathbf{x} \in P^c \cap D \cap \mathrm{IvIn}_{\mathbf{f}}(D).\, \mathbf{x} \in \Big(\mathrm{IvIn}_{\mathbf{f}}(P)\Big)^c$.

Necessary-Sufficient Condition for SCI. Given a PCCDS $(D, \mathbf{f})$ and a semi-algebraic template P, to encode the conditions in Theorem 8 as polynomial formulas, it is sufficient to show that $\mathrm{In}_{\mathbf{f}}(D)$, $\mathrm{In}_{\mathbf{f}}(P)$, $\mathrm{IvIn}_{\mathbf{f}}(D)$ and $\mathrm{IvIn}_{\mathbf{f}}(P)$ are all SASs if D and P are SASs, for which we have the following lemmas[4].

Lemma 1. *For any polynomial p and PVF $\mathbf{f}$,*

$$\mathrm{In}_{\mathbf{f}}(p > 0) = \psi^+(p, \mathbf{f}) \text{ and}$$
$$\mathrm{In}_{\mathbf{f}}(p \geq 0) = \psi_0^+(p, \mathbf{f}),$$

where

$$\psi^+(p, \mathbf{f}) \mathrel{\widehat{=}} \bigvee_{0 \leq i \leq N_{p,\mathbf{f}}} \psi^{(i)}(p, \mathbf{f}) \text{ with } \psi^{(i)}(p, \mathbf{f}) \mathrel{\widehat{=}} \Big(\bigwedge_{0 \leq j < i} L_{\mathbf{f}}^j p = 0 \Big) \wedge L_{\mathbf{f}}^i p > 0, \text{ and}$$
$$\psi_0^+(p, \mathbf{f}) \mathrel{\widehat{=}} \psi^+(p, \mathbf{f}) \vee \Big(\bigwedge_{0 \leq j \leq N_{p,\mathbf{f}}} L_{\mathbf{f}}^j p = 0 \Big).$$

Lemma 2. *For an SAS D defined by (9) and a PVF $\mathbf{f}$, we have*

$$\mathrm{In}_{\mathbf{f}}(D) = \bigvee_{m=1}^{M} \bigwedge_{l=1}^{L_m} \mathrm{In}_{\mathbf{f}}(p_{ml} \rhd 0).$$

Lemma 3. *For any polynomial p and PVF $\mathbf{f}$,*

$$\mathrm{IvIn}_{\mathbf{f}}(p > 0) = \varphi^+(p, \mathbf{f}) \text{ and}$$
$$\mathrm{IvIn}_{\mathbf{f}}(p \geq 0) = \varphi_0^+(p, \mathbf{f}),$$

where

$$\varphi^+(p, \mathbf{f}) \mathrel{\widehat{=}} \bigvee_{0 \leq i \leq N_{p,\mathbf{f}}} \varphi^{(i)}(p, \mathbf{f}) \text{ with } \varphi^{(i)}(p, \mathbf{f}) \mathrel{\widehat{=}} \Big(\bigwedge_{0 \leq j < i} L_{\mathbf{f}}^j p = 0 \Big) \wedge (-1)^i \cdot L_{\mathbf{f}}^i p > 0, \text{ and}$$
$$\varphi_0^+(p, \mathbf{f}) \mathrel{\widehat{=}} \varphi^+(p, \mathbf{f}) \vee \Big(\bigwedge_{0 \leq j \leq N_{p,\mathbf{f}}} L_{\mathbf{f}}^j p = 0 \Big).$$

Lemma 4. *For an SAS D defined by (9) and a PVF $\mathbf{f}$, we have*

$$\mathrm{IvIn}_{\mathbf{f}}(D) = \bigvee_{m=1}^{M} \bigwedge_{l=1}^{L_m} \mathrm{IvIn}_{\mathbf{f}}(p_{ml} \rhd 0).$$

[4] In the presentation below, we adopt the convention that $\bigvee_{i \in \emptyset} \eta_i = \mathit{false}$ and $\bigwedge_{i \in \emptyset} \eta_i = \mathit{true}$, where η_i is a logical formula.

Now the main result on automatic SCI generation can be stated as follows.

Theorem 9 (Main Result). *A semi-algebraic template* $P(\mathbf{u},\mathbf{x})$ *defined by*

$$\bigvee_{k=1}^{K}\left(\bigwedge_{j=1}^{j_k} p_{kj}(\mathbf{u}_{kj},\mathbf{x}) \geq 0 \quad \wedge \bigwedge_{j=j_k+1}^{J_k} p_{kj}(\mathbf{u}_{kj},\mathbf{x}) > 0\right)$$

is a CI of the PCCDS $\big(D,\mathbf{f}\big)$ *with*

$$D \mathrel{\widehat{=}} \bigvee_{m=1}^{M}\left(\bigwedge_{l=1}^{l_m} p_{ml}(\mathbf{x}) \geq 0 \quad \wedge \bigwedge_{l=l_m+1}^{L_m} p_{ml}(\mathbf{x}) > 0\right),$$

if and only if $\mathbf{u}$ *satisfies*

$$\forall \mathbf{x}.\Big(\big(P \wedge D \wedge \Phi_D \rightarrow \Phi_P\big) \wedge \big(\neg P \wedge D \wedge \Phi_D^{\mathrm{Iv}} \rightarrow \neg \Phi_P^{\mathrm{Iv}}\big)\Big),$$

where

$$\Phi_D \mathrel{\widehat{=}} \bigvee_{m=1}^{M}\left(\bigwedge_{l=1}^{l_m} \psi_0^{+}(p_{ml},\mathbf{f}) \wedge \bigwedge_{l=l_m+1}^{L_m} \psi^{+}(p_{ml},\mathbf{f})\right),$$

$$\Phi_P \mathrel{\widehat{=}} \bigvee_{k=1}^{K}\left(\bigwedge_{j=1}^{j_k} \psi_0^{+}(p_{kj},\mathbf{f}) \wedge \bigwedge_{j=j_k+1}^{J_k} \psi^{+}(p_{kj},\mathbf{f})\right),$$

$$\Phi_D^{\mathrm{Iv}} \mathrel{\widehat{=}} \bigvee_{m=1}^{M}\left(\bigwedge_{l=1}^{l_m} \varphi_0^{+}(p_{ml},\mathbf{f}) \wedge \bigwedge_{l=l_m+1}^{L_m} \varphi^{+}(p_{ml},\mathbf{f})\right),$$

$$\Phi_P^{\mathrm{Iv}} \mathrel{\widehat{=}} \bigvee_{k=1}^{K}\left(\bigwedge_{j=1}^{j_k} \varphi_0^{+}(p_{kj},\mathbf{f}) \wedge \bigwedge_{j=j_k+1}^{J_k} \varphi^{+}(p_{kj},\mathbf{f})\right),$$

with $\psi^{+}(p,\mathbf{f}), \psi_0^{+}(p,\mathbf{f}), \varphi^{+}(p,\mathbf{f}), \varphi_0^{+}(p,\mathbf{f})$ *defined in Lemma 1 and 3 respectively.*

Please refer to [54] for the proofs of the above results.

Example 5. Let $\mathbf{f}(x,y) = (-2y, x^2)$ and $D \mathrel{\widehat{=}} \mathbb{R}^2$. Take a template: $P(\mathbf{u},\mathbf{x}) \mathrel{\widehat{=}} x - a \geq 0 \vee y - b > 0$ with $\mathbf{u} = (a,b)$. By Theorem 9, P is an SCI of $(D,\mathbf{f})$ if and only if a, b satisfy

$$\forall x \forall y.\Big((P \rightarrow \zeta) \wedge (\neg P \rightarrow \neg \xi)\Big),^{5}$$

[5] Note that in Theorem 9 φ_D and φ_D^{Iv} are trivially *true* when D equals $\mathbb{R}^n$.

where

$$\begin{aligned}\zeta \widehat{=} & (x-a>0) \vee (x-a=0 \wedge -2y>0) \\ & \vee (x-a=0 \wedge -2y=0 \wedge -2x^2 \geq 0) \\ & \vee (y-b>0) \vee (y-b=0 \wedge x^2>0) \\ & \vee (y-b=0 \wedge x^2=0 \wedge -4yx>0) \\ & \vee (y-b=0 \wedge x^2=0 \wedge -4yx=0 \wedge 8y^2-4x^3>0) \\ \xi \widehat{=} & (x-a>0) \vee (x-a=0 \wedge -2y<0) \\ & \vee (x-a=0 \wedge -2y=0 \wedge -2x^2 \geq 0) \\ & \vee (y-b>0) \vee (y-b=0 \wedge x^2<0) \\ & \vee (y-b=0 \wedge x^2=0 \wedge -4yx>0) \\ & \vee (y-b=0 \wedge x^2=0 \wedge -4yx=0 \wedge 8y^2-4x^3<0)\end{aligned}$$

In addition, we require the set $x+y \geq 0$ to be contained in P. By applying QE, we get $a+b \leq 0 \wedge b \leq 0$. Let $a=-1$ and $b=-0.5$, and we obtain an SCI $P \widehat{=} x+1 \geq 0 \vee y+0.5>0$, which is shown in IV of Figure 2.

3.6 SGI Generation

Now the method for generating SGIs for a PHA $\mathcal{H} \widehat{=} (Q, X, f, D, E, G, R, \Xi)$ can be stated as the following steps.

I. Predefine a familiy of semi-algebraic templates $I_q(\mathbf{u}, \mathbf{x})$[6] with degree bound d for each $q \in Q$, as the SCI to be generated at mode q.

II. Translate conditions for the family of $I_q(\mathbf{u}, \mathbf{x})$ to be a GI of $\mathcal{H}$, i.e.
   - $\Xi_q \subseteq I_q$ for all $q \in Q$;
   - for any $e=(q,q') \in E$, if $\mathbf{x} \in I_q \cap G_e$, then $\mathbf{x}'=R_e(\mathbf{x}) \in I_{q'}$;
   - for any $q \in Q$, I_q is a CI of $(D_q, \mathbf{f}_q)$

   into a set of first-order real arithmetic formulas, i.e.

   (1) $\forall \mathbf{x}.\big(\Xi_q \rightarrow I_q(\mathbf{u}, \mathbf{x})\big)$ for all $q \in Q$;

   (2) $\forall \mathbf{x}, \mathbf{x}'.\big(I_q(\mathbf{u}, \mathbf{x}) \wedge G_e \wedge \mathbf{x}' = R_e(\mathbf{x}) \rightarrow I_{q'}(\mathbf{u}, \mathbf{x}')\big)$ for all $q \in Q$ and all $e=(q,q') \in E$, where $\mathbf{x}'$ is a vector of new variables with the same dimension as $\mathbf{x}$, and $I_{q'}(\mathbf{u}, \mathbf{x}')$ is obtained by substituting $\mathbf{x}'$ for $\mathbf{x}$ in $I_{q'}(\mathbf{u}, \mathbf{x})$;

   (3) $\forall \mathbf{x}.\Big(\big(I_q(\mathbf{u}, \mathbf{x}) \wedge D_q \wedge \Phi_{D_q} \rightarrow \Phi_{I_q}\big) \wedge \big(\neg I_q(\mathbf{u}, \mathbf{x}) \wedge D_q \wedge \Phi_{D_q}^{\mathrm{Iv}} \rightarrow \neg \Phi_{I_q}^{\mathrm{Iv}}\big)\Big)$ for each $q \in Q$, as defined in Theorem 9.

   Regarding the verification of a safety property $\mathcal{S}$, there may be a fourth set of formulas:

   (4) $\forall \mathbf{x}.(I_q(\mathbf{u}, \mathbf{x}) \longrightarrow S_q)$ for all $q \in Q$.

[6] Templates at different modes have different sets of parameters. Here we simply collect all the parameters together into a w-tuple $\mathbf{u}$.

III. Take the conjunction of all the formulas in Step 2 and apply QE to get a QFF $\phi(\mathbf{u})$. Then choose a specific $\mathbf{u}_0$ from $\phi(\mathbf{u})$ with a tool like Z3 [26], and the set of instantiations $I_{q,\mathbf{u}_0}(\mathbf{x})$ form a GI of $\mathcal{H}$.

The above method is *relatively complete* with respect to the predefined set of templates, that is, if there exist SGIs in the form of the predefined templates then we are able to find one.

Example 6. The Thermostat example taken from [6] can be described by the HA in Fig. 3. The system has three modes: Cool (q_{cl}), Heat (q_{ht}) and Check (q_{ck}); and 2 continuous variables: temperature T and timer clock c. All the domains, guards, reset functions and continuous dynamics are included in Fig. 3. We want to verify that under the initial condition $\Xi_{\mathcal{H}} \widehat{=} \{q_{\mathrm{ht}}\} \times X_0$ with $X_0 \widehat{=} c = 0 \wedge 5 \leq T \leq 10$, the safety property $S \widehat{=} T \geq 4.5$ is satisfied at all modes.

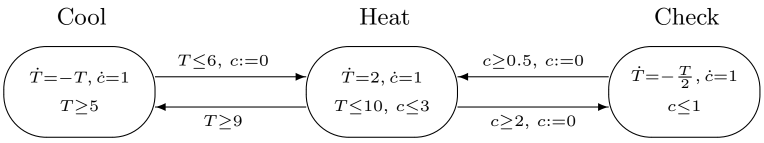


Fig. 3. A hybrid automaton describing the Thermostat system

Using the above SGI generation method, the following set of templates are predefined:

- $I_{q_{\mathrm{ht}}} \widehat{=} T + a_1 c + a_0 \geq 0 \wedge c \geq 0$;
- $I_{q_{\mathrm{cl}}} \widehat{=} T + a_2 \geq 0$;
- $I_{q_{\mathrm{ck}}} \widehat{=} T \geq a_3 c^2 - 4.5c + 9 \wedge c \geq 0 \wedge c \leq 1$

with indeterminates a_0, a_1, a_2 and a_3. By deriving verification conditions and applying QE we get the following constraint on a_0, a_1, a_2, a_3:

$$10a_3 - 9 \leq 0 \wedge 2a_3 - 1 \geq 0 \wedge a_1 + 2 = 0 \wedge a_0 + 2a_1 + 9 = 0 \wedge a_2 - a_0 = 0\,.$$

By choosing $a_0 = -5, a_1 = -2, a_2 = -5, a_3 = \frac{1}{2}$, the following SGI instantiation is obtained

- $I_{q_{\mathrm{ht}}} \widehat{=} T \geq 2c + 5 \wedge c \geq 0$;
- $I_{q_{\mathrm{cl}}} \widehat{=} T \geq 5$;
- $I_{q_{\mathrm{ck}}} \widehat{=} 2T \geq c^2 - 9c + 18 \wedge c \geq 0 \wedge c \leq 1$,

and the safety property is successfully verified.

4 Switching Controller Synthesis

4.1 Problem Description

In verification problems, a given hybrid system is proved to satisfy a desired safety (or other) property. A *synthesis* problem is harder given that the focus is on *designing* a hybrid system that will satisfy a safety requirement, reach a given set of states, or meet an optimality criterion, or a desired combination of these requirements.

In this section we talk about the synthesis of *switching controllers* for hybrid systems with safety requirements. That is, given a hybrid system and a safety requirement, we aim to identify a subset of continuous states from each original transition guard, such that if only at these states is mode switching allowed, then the system can run forever without violating the required safety property.

The formal definition of the switching controller synthesis problem w.r.t. safety requirement can be given in the way of [10]. Note that the specification of hybrid automata has been simplified by assuming that the initial condition is identical with the domain, and all reset functions are identity mappings.

Problem 1 (Switching Controller Synthesis for Safety). Given a hybrid automaton $\mathcal{H} = (Q, X, f, D, E, G)$ and a safety property $\mathcal{S}$, find a hybrid automaton $\mathcal{H}' = (Q, X, f, D', E, G')$ such that

(r1) Refinement: for any $q \in Q$, $D'_q \subseteq D_q$, and for any $e \in E$, $G'_e \subseteq G_e$;
(r2) Safety: for any $(q, \mathbf{x}) \in \mathcal{R}_{\mathcal{H}'}$, $\mathbf{x} \in S_q$;
(r3) Non-blocking: $\mathcal{H}'$ is non-blocking.

If such $\mathcal{H}'$ exists, then $\mathcal{SC} \mathrel{\widehat{=}} \{G'_e \subseteq \mathbb{R}^n \mid e \in E\}$ is a *switching controller* satisfying the safety requirement $\mathcal{S}$, and $D_{\mathcal{H}'} \mathrel{\widehat{=}} \bigcup_{q \in Q}(\{q\} \times D'_q)$ is the *controlled invariant set* rendered by $\mathcal{SC}$.

In the following, the theory and techniques on continuous invariant generation developed in Section 3 will be exploited to solve Problem 1.

4.2 A Synthesis Procedure Based on CI Generation

To solve Problem 1 amounts to refining the domains and guards of $\mathcal{H}$ by removing so-called *bad* states. A state $(q, \mathbf{x}) \in D_{\mathcal{H}}$ is *bad* if the hybrid trajectory starting from $(q, \mathbf{x})$ either blocks $\mathcal{H}$ or violates $\mathcal{S}$; otherwise if the trajectory starting from $(q, \mathbf{x})$ can either be extended to infinite time or execute infinitely many discrete transitions while maintaining $\mathcal{S}$, then $(q, \mathbf{x})$ is called a *good* state. By Definition 13, the set of good states of $\mathcal{H}$ can be approximated appropriately using CIs, which results in the following solution to Problem 1.

Theorem 10. *Let $\mathcal{H}$ and $\mathcal{S}$ be the same as in Problem 1. Suppose D'_q is a closed subset of $\mathbb{R}^n$ for all $q \in Q$ and $\bigcup_{q \in Q} D'_q$ is non-empty. If we have*

(c1) for all $q \in Q$, $D'_q \subseteq D_q \cap S_q$; and

(c2) for all $q \in Q$, D'_q is a CI of $(H_q, \mathbf{f}_q)$, where

$$H_q \mathrel{\widehat{=}} \Big(\bigcup_{e=(q,q')\in E} G'_e \Big)^c \quad \textit{with} \quad G'_e \mathrel{\widehat{=}} G_e \cap D'_{q'} \,,$$

then the HA $\mathcal{H}' = (Q, X, f, D', E, G')$ is a solution to Problem 1.

Please refer to [49] for the proof of this theorem.

Intuitively, by (c1), D'_q is a refinement of D_q and is also contained in the safe region S_q, thus guaranteeing (r1) and (r2) of Problem 1; by (c2), any trajectory starting from D'_q will either stay in D'_q forever[7], or finally intersect one of the transition guards enabling jumps from q to a certain q', thus guaranteeing (r3) of Problem 1.

Based on Theorem 10, the following template-based method for synthesizing switching controllers for PHA with semi-algebraic safety requirement is proposed, by incorporating the automatic SCI generation method in Section 3.4 and 3.5.

(s1) **Template Assignment:** assign to each $q \in Q$ a semi-algebraic template specifying D'_q, which will be required (see step (s3)) to be a refinement of D_q, as well as the CI to be generated at mode q;
(s2) **Guard Refinement:** refine guard G_e for each $e = (q, q') \in E$ by setting $G'_e \mathrel{\widehat{=}} G_e \cap D'_{q'}$;
(s3) **Deriving Synthesis conditions:** encode (c1) and (c2) in Theorem 10 into first-order polynomial formulas; the encoding of condition (c1) is straightforward, while encoding of (c2) is based on Theorem 9;
(s4) **Constraint Solving:** apply QE to the fisrt-order formulas derived in (s3) and a QFF will be returned specifying the set of all possible values for the parameters appearing in templates;
(s5) **Parameters Instantiation:** a switching controller can be obtained by an appropriate instantiation of D'_q and G'_e such that D'_q are closed sets for all $q \in Q$, and D'_q is non-empty for at least one $q \in Q$; if such an instantiation is not found, we choose a new set of templates and go back to (s1).

In the above procedure, the method for SCI generation based on a necessary and sufficient criterion for SCIs is used as an integral component. As a result, the above controller synthesis method is relatively complete with respect to a given family of templates, thus having more possibility of discovering a switching controller.

The shape of chosen templates in (s1) determines the likelihood of success of the above procedure, as well as the complexity of QE in (s4). Next, heuristics for choosing appropriate templates will be discussed using the *qualitative analysis* proposed in [47].

[7] Actually in Theorem 10, for any mode $q \in Q$, $\mathbf{f}_q$ is required to be a *complete* vector field, that is, for any $\mathbf{x}_0 \in \mathbb{R}^n$, the solution $\mathbf{x}(\mathbf{x}_0; t)$ to $\dot{\mathbf{x}} = \mathbf{f}_q$ exists on $[0, \infty)$.

4.3 Heuristics for Predefining Templates

The key steps of the qualitative analysis used in [47] are as follows.

1. The evolution behavior (increasing or decreasing) of continuous state variables in each mode is inferred from the differential equations (using first or second order derivatives);
2. *control critical* modes, at which the maximal (or minimal) value of a continuous state variable is achieved, can be identified;
3. the safety requirement is imposed to obtain constraints on guards of transitions leading to control critical modes, and
4. then this information on transition guards is propagated to other modes.

Next, we illustrate how such an analysis helps in predefining templates for a nuclear reactor temperature control system discussed in [47].

Example 7. The nuclear reactor system consists of a reactor core and a cooling rod which is immersed into and removed out of the core periodically to keep the temperature of the core, denoted by x, in a certain range. Denote the fraction of the rod immersed into the reactor by p. Then the initial specification of this system can be represented using the hybrid automaton in Fig. 4. The goal is to synthesize a switching controller for this system with the global safety requirement that the temperature of the core lies between 510 and 550, i.e. $S_i \widehat{=} 510 \leq x \leq 550$ for $i = 1, 2, 3, 4$.

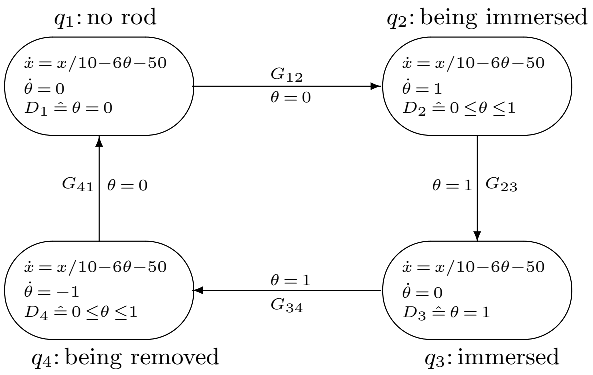

q_1: no rod
$\dot{x} = x/10-6\theta-50$
$\dot{\theta} = 0$
$D_1 \widehat{=} \theta = 0$
G_{12}
$\theta = 0$
q_2: being immersed
$\dot{x} = x/10-6\theta-50$
$\dot{\theta} = 1$
$D_2 \widehat{=} 0 \leq \theta \leq 1$
G_{41}
$\theta = 0$
$\theta = 1$
G_{23}
$\dot{x} = x/10-6\theta-50$
$\dot{\theta} = -1$
$D_4 \widehat{=} 0 \leq \theta \leq 1$
$\theta = 1$
G_{34}
$\dot{x} = x/10-6\theta-50$
$\dot{\theta} = 0$
$D_3 \widehat{=} \theta = 1$
q_4: being removed
q_3: immersed

Fig. 4. Nuclear reactor temperature control

1) **Refine Domains.** Using the safety requirement, domains D_i for $i = 1, 2, 3, 4$ are refined by $D_i^s \widehat{=} D_i \cap S_i$, e.g. $D_1^s \widehat{=} \theta = 0 \wedge 510 \leq x \leq 550$.

2) **Infer Continuous Evolutions.** Let $l_1 \widehat{=} x/10 - 6\theta - 50 = 0$ be the *zero-level* set of $\dot{x}$ and check how x and θ evolve in each mode. For example, in D_2^s, $\dot{x} > 0$ on the left of l_1 and $\dot{x} < 0$ on the right; since θ increases from 0 to 1, x first increases then decreases and achieves maximal value when crossing l_1. (See Fig. 5.)
3) **Identify Critical Control Modes.** By 2), q_2 and q_4 are critical control modes at which the evolution direction of x changes, and thus maximal (or minimal) value of x is achieved.
4) **Generate Control Points.** By 3), we can get a control point $E(5/6, 550)$ at q_2 by taking the intersection of l_1 and the safety upper bound $x = 550$; and $F(1/6, 510)$ at q_4 is obtained by taking the intersection of l_1 and the safety lower bound $x = 510$.
5) **Propagate Control Points.** E is backward propagated to $A(0, a)$ using the trajectory $\widehat{AE}$ through E defined by $\mathbf{f}_{q_2}$, and then to $C(1, c)$ using the trajectory $\widehat{CA}$ through A defined by $\mathbf{f}_{q_4}$; similarly, by propagating F we get D and B.
6) **Construct Templates.** For brevity, we only show how to construct D_2'. Intuitively, $\theta = 0$, $\theta = 1$, $\widehat{AE}$ and $\widehat{BD}$ form the boundaries of D_2'. In order to get a semi-algebraic template, we need to fit $\widehat{AE}$ and $\widehat{BD}$ (which are generally not polynomial curves) by polynomials using points A, E and B, D respectively. By the inference of 2), $\widehat{AE}$ has only one extreme point (also the maximum point) E in D_2^s, and is tangential to $x = 550$ at E. A simple algebraic curve that can exhibit a shape similar to $\widehat{AE}$ is the parabola through A, E opening downward with $l_2 \widehat{=} \theta = \frac{5}{6}$ the axis of symmetry. Therefore to minimize the degree of terms appearing in templates, we do not resort to polynomials with degree greater than 2. This parabola can be computed using the coordinates of A, E as: $x - 550 - \frac{36}{25}(a - 550)(\theta - \frac{5}{6})^2 = 0$, with a the parameter to be determined.

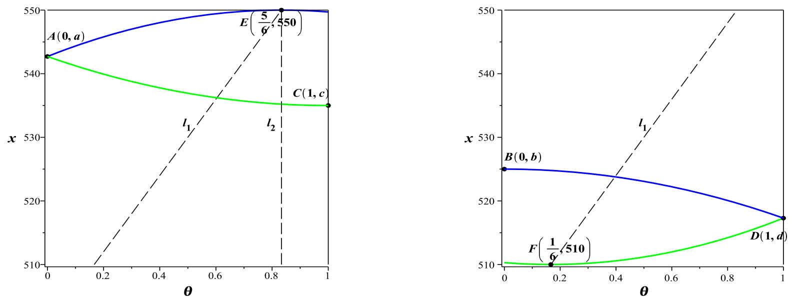


Fig. 5. Control points propagation

Through the above analysis, we generate the following templates:

- $D'_1 \mathrel{\widehat{=}} \theta = 0 \wedge 510 \le x \le a\,;$
- $D'_2 \mathrel{\widehat{=}} 0 \le \theta \le 1 \wedge x - b \ge \theta(d-b) \wedge x - 550 - \frac{36}{25}(a-550)(\theta - \frac{5}{6})^2 \le 0\,;$
- $D'_3 \mathrel{\widehat{=}} \theta = 1 \wedge d \le x \le 550\,;$
- $D'_4 \mathrel{\widehat{=}} 0 \le \theta \le 1 \wedge x - a \le \theta(c-a) \wedge x - 510 - \frac{36}{25}(d-510)(\theta - \frac{1}{6})^2 \ge 0\,,$

in which a, b, c, d are parameters satisfying

$$510 \le b \le a \le 550 \wedge 510 \le d \le c \le 550\,.$$

Note that without qualitative analysis, a single generic *quadratic* polynomial over θ and x would require $\binom{2+2}{2} = 6$ parameters.

Based on the synthesis procedure (s1)–(s5) presented in Section 4.2, we show below how to synthesize a switching controller for the system in Example 7 step by step.

(s1) The four templates are defined as the above D'_i for $1 \le i \le 4$.

(s2) The four guards are refined by $G'_{ij} \mathrel{\widehat{=}} G_{ij} \cap D'_j$ and then simplified to:
- $G'_{12} \mathrel{\widehat{=}} \theta = 0 \wedge b \le x \le a\,;$
- $G'_{23} \mathrel{\widehat{=}} \theta = 1 \wedge d \le x \le 550\,;$
- $G'_{34} \mathrel{\widehat{=}} \theta = 1 \wedge d \le x \le c\,;$
- $G'_{41} \mathrel{\widehat{=}} \theta = 0 \wedge 510 \le x \le a\,.$

(s3) The derived synthesis condition, which is a first-order polynomial formula in the form of $\phi \mathrel{\widehat{=}} \forall x \forall \theta . \varphi(a, b, c, d, x, \theta)$, is not included here due to its big size.

(s4) By applying QE to ϕ we get the following sample solution to the parameters:

$$a = \frac{6575}{12} \wedge b = \frac{4135}{8} \wedge c = \frac{4345}{8} \wedge d = \frac{6145}{12}\ . \tag{10}$$

(s5) Instantiate D'_i and G'_{ij} by (10). It is obvious that all D'_i are nonempty closed[8] sets. According to Theorem 9, we get a switching controller guaranteeing safety property for the nuclear reactor system, i.e.
- $G'_{12} \mathrel{\widehat{=}} \theta = 0 \wedge 4135/8 \le x \le 6575/12\,;$
- $G'_{23} \mathrel{\widehat{=}} \theta = 1 \wedge 6145/12 \le x \le 550\,;$
- $G'_{34} \mathrel{\widehat{=}} \theta = 1 \wedge 6145/12 \le x \le 4345/8\,;$
- $G'_{41} \mathrel{\widehat{=}} \theta = 0 \wedge 510 \le x \le 6575/12\,.$

In [47], an upper bound $x = 547.97$ for G_{12} and a lower bound $x = 512.03$ for G_{34} are obtained by solving the differential equations at mode q_2 and q_4 respectively. By (10), the corresponding bounds generated here are $x \le \frac{6575}{12} = 547.92$ and $x \ge \frac{6145}{12} = 512.08$.

As should be evident from the above discussion, in contrast to [47], where differential equations are solved to get closed-form solutions, here good approximate results are obtained without requiring closed-form solutions. This indicates that the controller synthesis approach based on CI generation should work well for hybrid automata where differential equations for modes need not have closed form solutions.

[8] Actually all D'_i become closed sets naturally by the construction of templates, in which only $\ge, \le, =$ relations appear conjunctively.

4.4 Synthesis of Optimal Controllers

Most of the discussion so far on switching controller synthesis is based on meeting the safety requirements. As a result, there is still considerable flexibility left in designing controllers to meet other objectives. One important criterion for further refinement of controllers is *optimality*, i.e. to optimize a *reward/penalty* function that reflects the performance of the controlled system.

The optimal switching controller synthesis problem studied in this section can be stated as follows.

Problem 2. Suppose $\mathcal{H}$ is a hybrid automaton whose transition guards are not determined but specified by a vector of parameters $\mathbf{u}$. Associated with $\mathcal{H}$ is an *objective function* g in $\mathbf{u}$. The task is to determine values of $\mathbf{u}$, or a relation over $\mathbf{u}$, such that $\mathcal{H}$ can take discrete jumps at desired conditions, thus guaranteeing

1) a safety requirement $\mathcal{S}$ is satisfied; and
2) an optimization goal $\mathcal{G}$, possibly

$$\min_{\mathbf{u}} g(\mathbf{u}),\ \max_{\mathbf{u}_2}\min_{\mathbf{u}_1} g(\mathbf{u}),\text{ or }\min_{\mathbf{u}_3}\max_{\mathbf{u}_2}\min_{\mathbf{u}_1} g(\mathbf{u})^{9}\ \text{ is achieved.}$$

The determined values of $\mathbf{u}$ or relations over $\mathbf{u}$ are called the *optimal switching controller.*

If $\mathcal{H}$ is a PHA and $\mathcal{S}$ is a semi-algebraic safety property, then Problem 2 can be solved by following the steps (s1)–(s4) in Section 4.2 and then solving an optimization problem with objective $\mathcal{G}$. In particular, if g is a polynomial function, then the optimization problem can also be encoded into first-order polynomial formulas and then solved by QE.

In detail, the approach for solving the optimal controller synthesis problem can be described as the following steps.

Step 1. *Derive constraint $\mathcal{D}(\mathbf{u})$ on $\mathbf{u}$ from the safety requirements of the system.*

The reachable set $R_{\mathcal{H}}$ (parameterized by $\mathbf{u}$) is either computed exactly, or approximated using SCIs (with $\mathbf{u}$ and possibly others as parameters). Then the safety requirement $\mathcal{S}$ is imposed to derive constraint on $\mathbf{u}$ using QE.

Step 2. *Encode the optimization problem $\mathcal{G}$ over constraint $\mathcal{D}(\mathbf{u})$ into a quantified first-order polynomial formula $\mathbf{Qu}.\varphi(\mathbf{u}, z)$, where z is a fresh variable.*

The encoding is based on the following proposition, in which all the aforementioned optimization objectives are discussed together.

Proposition 4. *Suppose $g_1(\mathbf{u}_1)$, $g_2(\mathbf{u}_1, \mathbf{u}_2)$, $g_3(\mathbf{u}_1, \mathbf{u}_2, \mathbf{u}_3)$ are polynomials, and $\mathcal{D}_1(\mathbf{u}_1)$, $\mathcal{D}_2(\mathbf{u}_1, \mathbf{u}_2)$, $\mathcal{D}_3(\mathbf{u}_1, \mathbf{u}_2, \mathbf{u}_3)$ are nonempty compact (i.e. bounded closed) SASs. Then there exist c_1, c_2, $c_3 \in \mathbb{R}$ s.t.*

[9] The elements of $\mathbf{u}$ are divided into groups $\mathbf{u}_1, \mathbf{u}_2, \mathbf{u}_3, \dots$ according to their roles in $\mathcal{G}$.

$$\exists \mathbf{u}_1.(\mathcal{D}_1 \wedge g_1 \le z) \iff z \ge c_1\,, \tag{11}$$

$$\forall \mathbf{u}_2.\Big(\exists \mathbf{u}_1.\mathcal{D}_2 \longrightarrow \exists \mathbf{u}_1.(\mathcal{D}_2 \wedge g_2 \le z)\Big) \iff z \ge c_2\,, \tag{12}$$

$$\exists \mathbf{u}_3.\Big((\exists \mathbf{u}_1\mathbf{u}_2.\mathcal{D}_3) \wedge \forall \mathbf{u}_2.\Big(\exists \mathbf{u}_1.\mathcal{D}_3 \longrightarrow \exists \mathbf{u}_1.(\mathcal{D}_3 \wedge g_3 \le z)\Big)\Big) \iff z \rhd c_3, \tag{13}$$

where $\rhd \in \{>, \ge\}$*, and* c_1, c_2, c_3 *satisfy*

$$\begin{aligned} c_1 &= \min_{\mathbf{u}_1} g_1(\mathbf{u}_1) \quad \text{over}\, \mathcal{D}_1(\mathbf{u}_1)\,, & (14)\\ c_2 &= \sup_{\mathbf{u}_2} \min_{\mathbf{u}_1} g_2(\mathbf{u}_1, \mathbf{u}_2) \quad \text{over}\, \mathcal{D}_2(\mathbf{u}_1, \mathbf{u}_2)\,, & (15)\\ c_3 &= \inf_{\mathbf{u}_3} \sup_{\mathbf{u}_2} \min_{\mathbf{u}_1} g_3(\mathbf{u}_1, \mathbf{u}_2, \mathbf{u}_3) \quad \text{over}\, \mathcal{D}_3(\mathbf{u}_1, \mathbf{u}_2, \mathbf{u}_3)\,. & (16) \end{aligned}$$

The proof of this proposition can be found in [95].

Step 3. *Apply QE to* $\mathbf{Qu}.\varphi(\mathbf{u}, z)$ *and from the result we can retrieve the optimal value of* $\mathcal{G}$ *and the corresponding optimal controller* $\mathbf{u}$*.*

Using the above procedure, the issues of synthesis, verification and optimization for hybrid systems are integrated into one elegant framework. Compared to numerical approaches, using the QE-based method, the synthesized controllers are guaranteed to be correct and better optimal optimal values can be obtained.

4.5 Oil Pump: A Case Study

We illustrate the above approach on an industrial oil pump example studied in [18].

The whole system consists of a machine, an accumulator, a reservoir and a pump. The machine consumes oil periodically out of the accumulator with a period of $20s$ (second) for one consumption cycle. The profile of consumption rate is shown in Fig. 6. The pump adds oil from the reservoir into the accumulator with power $2.2l/s$ (liter/second). There is an additional physical constraint requiring a *latency* of at least $2s$ between any two consecutive operations of the pump.

Control objective for this system is to switch on/off the pump at appropriate time points

$$0 \le t_1 \le t_2 \le \cdots \le t_k \le t_{k+1} \le \cdots \tag{17}$$

in order to

1) maintain the oil volume $v(t)$ in the accumulator within a safe range $[V_{\min}, V_{\max}]$ at any time, where $V_{\min} = 4.9l$, $V_{\max} = 25.1l$; and
2) minimize the average accumulated oil volume in one cycle, i.e. $\frac{1}{T}\int_{t=0}^{T} v(t)$.

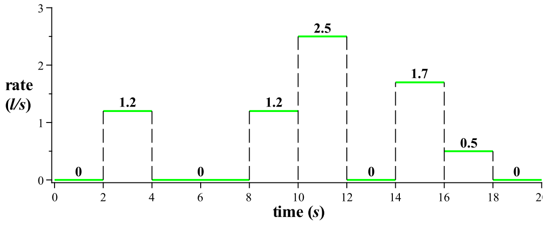


Fig. 6. Consumption rate in each cycle

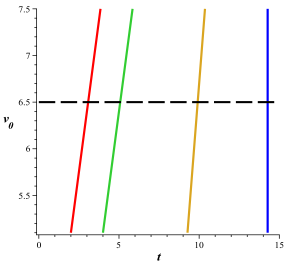


Fig. 7. Optimal switching controller for the oil pump

The second objective is important because the average oil level reflects the energy cost of the system.

Following [18], the time points to switch on/off the pump in one consumption cycle is determined by measuring the oil volume v_0 at the beginning of each cycle. Besides, it is assumed that the pump is operated (turned on/off) at most 4 times in one cycle.

The system along with the safety and optimality requirements can all be exactly modeled by first-order polynomial formulas. By applying various QE heuristics, the following results are obtained:

- The optimal switching controller is

$$t_1 = \frac{10v_0 - 25}{13} \wedge t_2 = \frac{10v_0 + 1}{13} \wedge t_3 = \frac{10v_0 + 153}{22} \wedge t_4 = \frac{157}{11} \ , \tag{18}$$

  where t_1, t_2, t_3, t_4 are the 4 time points to operate the pump in one cycle, and $v_0 \in [5.1, 7.5]$ is the measurement of the initial oil volume at the beginning of each cycle. If $v_0 = 6.5$, then by (18) the pump should be switched on at $t_1 = 40/13$, off at $t_2 = 66/13$, then on at $t_3 = 109/11$, and finally off at $t_4 = 157/11$ (dashed line in Fig. 7).
- The optimal average accumulated oil volume obtained using the strategy given by (18) is $V_{opt} = \frac{215273}{28600} = 7.53$, which is a significant improvement (over 5%) compared to the optimal value 7.95 reported in [18]. If the pump is allowed to be turned on more times, then even better controllers can be generated ($V_{opt} = 7.35$ if the pump is allowed to be turned on at most 3 times in one cycle).

More details about this case study can be found in [94,95].

5 Hybrid CSP

HCSP [37,98], which extends CSP by introducing differential equations for modelling continuous evolutions and interrupts, is a formal language for describing hybrid systems. In HCSP, exchange of data among processes is described solely by communications; no shared variable is allowed between different processes in parallel, so each process variable is local to the respective sequential component. We denote by $\mathcal{V}$ ranged over $x, y, s, \ldots$ the set of variables, and by Σ ranged over $ch, ch_1, \ldots$ the set of channels. The syntax of HCSP is given as follows:

$$\begin{aligned}
P ::= &\ \mathbf{skip} \mid x := e \mid \text{wait } d \mid ch?x \mid ch!e \mid P;Q \mid B \to P \mid P \sqcup Q \mid [\!]_{i\in I}(ch_i* \to Q_i) \\
&\mid P^* \mid \langle \mathcal{F}(\dot{s}, s) = 0 \& B\rangle \mid \langle \mathcal{F}(\dot{s}, s) = 0 \& B\rangle \trianglerighteq_d Q \\
&\mid \langle \mathcal{F}(\dot{s}, s) = 0 \& B\rangle \trianglerighteq [\!]_{i\in I}(ch_i* \to Q_i) \\
S ::= &\ P \mid S \| S
\end{aligned}$$

Here $ch, ch_i \in \Sigma$, ch_i* stands for a communication event, i.e., either $ch_i?x$ or $ch_i!e$, $x, s \in \mathcal{V}$, B and e are Boolean and arithmetic expressions, d is a non-negative real constant, P, Q, Q_i are sequential processes, and S stands for a system, i.e., an HCSP process.

The intended meaning of the individual constructs is as follows:

- **skip** terminates immediately having no effect on variables.
- $x := e$ assigns the value of expression e to x and then terminates.
- wait d will keep idle for d time units keeping variables unchanged.
- $ch?x$ receives a value along channel ch and assigns it to x.
- $ch!e$ sends the value of e along channel ch. A communication takes place when both the sending and the receiving parties are ready, and may cause one side to wait.
- The sequential composition $P;Q$ behaves as P first, and if it terminates, as Q afterwards.
- The alternative $B \to P$ behaves as P if B is true; otherwise it terminates immediately.
- $P \sqcup Q$ denotes internal choice. It behaves as either P or Q, and the choice is made by the process.
- $[\!]_{i\in I}(ch_i* \to Q_i)$ denotes communication controlled external choice. I is supposed to be finite. As soon as one of the communications ch_i* takes place, the process continues as the respective guarded Q_i.
- The repetition P^* executes P for some finite number of times.
- $\langle \mathcal{F}(\dot{s}, s) = 0 \& B\rangle$ is the continuous evolution statement (hereafter shortly *continuous*). It forces the vector s of real variables to obey the differential equations $\mathcal{F}$ as long as the boolean expression B, which defines the *domain of* s, holds, and terminates when B turns false.
- $\langle \mathcal{F}(\dot{s}, s) = 0 \& B\rangle \trianglerighteq_d Q$ behaves like $\langle \mathcal{F}(\dot{s}, s) = 0 \& B\rangle$, if that continuous terminates before d time units. Otherwise, after d time units of evolution according to $\mathcal{F}$, it moves on to execute Q.

- $\langle \mathcal{F}(\dot{s}, s) = 0 \& B\rangle \trianglerighteq []_{i\in I}(ch_i* \to Q_i)$ behaves like $\langle \mathcal{F}(\dot{s}, s) = 0\&B\rangle$, except that the continuous evolution is preempted as soon as one of the communications ch_i* takes place, which is followed by the respective Q_i. Notice that, if the continuous part terminates before a communication from among $\{ch_i*\}_I$ occurs, then the process terminates without communicating.
- $S_1 \| S_2$ behaves as if S_1 and S_2 run independently except that all communications along the common channels connecting S_1 and S_2 are to be synchronized. The processes S_1 and S_2 in parallel can neither share variables, nor input or output channels.

Note that some primitives of CSP and timed CSP are derivable from the above syntax, e.g. $\textbf{stop} \overset{\text{def}}{=} t := 0; \langle \dot{t} = 1\& true\rangle$. Specifically, some of the constructs in the above syntax can be defined with other ones and thus are not primitive either, for instance

$$\begin{aligned} \text{wait } d &\overset{\text{def}}{=} t := 0; \langle \dot{t} = 1 \& t < d\rangle, \\ []_{i\in I}(ch_i* \to Q_i) &\overset{\text{def}}{=} \textbf{stop} \trianglerighteq []_{i\in I}(ch_i* \to Q_i), \\ \langle \mathcal{F}(\dot{s}, s) = 0\&B\rangle \trianglerighteq_d Q &\overset{\text{def}}{=} t := 0; \langle F(\dot{s}, s) = 0 \wedge \dot{t} = 1\& t < d \wedge B\rangle; t \geq d \to Q. \end{aligned}$$

Example 8. Consider the classic plant-controller example: A plant is sensed by a computer periodically (say every d time units), and receives a control (u) from the digit controller soon after the sensing. Thus, it can be modelled by the following HCSP process:

$$((\langle F(u, s, \dot{s}) = 0\& true\rangle \trianglerighteq (c_{p2c}!s \to \textbf{skip})); c_{c2p}?u)^* \parallel (\textbf{wait } d; c_{p2c}?x; c_{c2p}!e(x))^*$$

where $\langle F(u, s, \dot{s}) = 0\& true\rangle$ describes the behaviour of the plant. We refer this HCSP process as *PLC* hereafter.

In the sequel, we use $\mathcal{V}(P)$ to stand for the set of local variables and $\Sigma(P)$ for the set of channels of a process P.

5.1 Notations

In order to define the real-time behavior of HCSP processes, we use non-negative reals $\mathbb{R}^+$ to model time, and introduce a global clock *now* as a system variable to record the time in the execution of a process.

A *timed communication* is of the form $\langle ch.c, b\rangle$, where $ch \in \Sigma$, $c \in \mathbb{R}$ and $b \in \mathbb{R}^+$, representing that a communication along channel ch occurs at time b with value c transmitted. The set $\Sigma \times \mathbb{R} \times \mathbb{R}^+$ of all timed communications is denoted by $T\Sigma$. The set of all timed traces is

$$T\Sigma^*_{\leq} = \{\gamma \in T\Sigma^* \mid \text{ if } \langle ch_1.c_1, b_1\rangle \text{ precedes } \langle ch_2.c_2, b_2\rangle \text{ in } \gamma, \text{ then } b_1 \leq b_2\}.$$

If $X \subseteq \Sigma$, $\gamma \upharpoonright_X$ is the projection of γ onto X.

Given two timed traces γ_1, γ_2, and $X \subseteq \Sigma$, the *alphabetized parallel* of γ_1 and γ_2 over X, denoted by $\gamma_1 \underset{X}{\|} \gamma_2$, results in a set of timed traces, defined by:

$$\langle\rangle \underset{X}{\|} \langle\rangle \overset{\text{def}}{=} \langle\rangle, \qquad \langle\rangle \underset{X}{\|} \gamma \overset{\text{def}}{=} \gamma \underset{X}{\|} \langle\rangle$$

$$\langle ch.a, b\rangle \cdot \gamma \underset{X}{\|} \langle\rangle \overset{\text{def}}{=} \begin{cases} \langle ch.a, b\rangle \cdot (\gamma \underset{X}{\|} \langle\rangle) \text{ if } ch \notin X \\ \emptyset \quad \text{otherwise} \end{cases}$$

$$\langle ch_1.a, t_1\rangle \cdot \gamma_1' \underset{X}{\|} \langle ch_2.b, t_2\rangle \cdot \gamma_2'$$

$$\overset{\text{def}}{=} \begin{cases} \langle ch_1.a, t_1\rangle \cdot (\gamma_1' \underset{X}{\|} \gamma_2') \quad \text{if } ch_1 = ch_2 \in X, a = b, t_1 = t_2 \\ \langle ch_1.a, t_1\rangle \cdot (\gamma_1' \underset{X}{\|} (\langle ch_2.b, t_2\rangle \cdot \gamma_2')) \cup \langle ch_2.b, t_2\rangle \cdot ((\langle ch_1.a, t_1\rangle \cdot \gamma_1') \underset{X}{\|} \gamma_2') \\ \quad \text{otherwise if } ch_1, ch_2 \notin X, t_1 = t_2 \\ \langle ch_1.a, t_1\rangle \cdot (\gamma_1' \underset{X}{\|} (\langle ch_2.b, t_2\rangle \cdot \gamma_2')) \quad \text{otherwise if } ch_1 \notin X, t_1 \le t_2 \\ \langle ch_2.b, t_2\rangle \cdot ((\langle ch_1.a, t_1\rangle \cdot \gamma_1') \underset{X}{\|} \gamma_2') \quad \text{otherwise if } ch_2 \notin X, \text{ and } t_2 \le t_1 \\ \emptyset \quad \text{otherwise} \end{cases}$$

To model synchronization of communication events, we need to describe their readiness, and meanwhile, to record the timed trace of communications having occurred till now. Each *communication event* has the form of $\gamma.ch?$ or $\gamma.ch!$, to represent that $ch?$ (resp. $ch!$) is ready to occur, and before that the sequence of communications γ have occurred. Therefore, we introduce two system variables, rdy and tr, to represent the ready set of communication events and the timed communication trace accumulated, at each time point during process execution. In what follows, we use $\mathcal{V}^+(P)$ to denote $\mathcal{V}(P) \cup \{rdy, tr, now\}$.

For a process P, a state σ of P is an assignment to associate a value from the respective domain to each variable in $\mathcal{V}^+(P)$. Given two states σ_1 and σ_2, we say σ_1 and σ_2 are parallelable iff $Dom(\sigma_1) \cap Dom(\sigma_2) = \{rdy, tr, now\}$ and $\sigma_1(now) = \sigma_2(now)$. Paralleling them over $X \subseteq \Sigma$ results in a set of new states, denoted by $\sigma_1 \uplus \sigma_2$, any of which σ is given by

$$\sigma(v) \overset{\text{def}}{=} \begin{cases} \sigma_1(v) & \text{if } v \in Dom(\sigma_1) \setminus Dom(\sigma_2), \\ \sigma_2(v) & \text{if } v \in Dom(\sigma_2) \setminus Dom(\sigma_1), \\ \sigma_1(now) & \text{if } v = now, \\ \gamma, \text{ where } \gamma \in \sigma_1(tr) \underset{X}{\|} \sigma_2(tr) & \text{if } v = tr, \\ \sigma_1(rdy) \cup \sigma_2(rdy) & \text{if } v = rdy. \end{cases}$$

It makes no sense to distinguish any two states in $\sigma_1 \uplus \sigma_2$, so hereafter we abuse $\sigma_1 \uplus \sigma_2$ to represent any of its elements.

5.2 Operational Semantics

As mentioned above, we use *now* to record the time during process execution. A state, ranging over σ, σ_1, assigns respective value to each variable in $\mathcal{V}^+(P)$; moreover, we introduce *flow*, ranging over H, H_1, defined on a time interval, assigns a state to each point in the interval.

Each transition relation has the form of $(P, \sigma) \xrightarrow{\alpha} (P', \sigma', H)$, where P is a process, σ, σ' are states, H is a flow. It records that starting from initial state σ, P evolves into P' and ends in state σ' and flow H, while performing event α. When the transition is discrete and thus produces a flow on an interval that contains only one point, we will write $(P, \sigma) \xrightarrow{\alpha} (P', \sigma')$ instead of $(P, \sigma) \xrightarrow{\alpha} (P', \sigma', \{\sigma(now) \mapsto \sigma'\})$. The label α represents events, which can be an internal event like skip, assignment, or a termination of a continuous *etc*, uniformly denoted by τ, or an external communication event $ch!c$ or $ch?c$, or an internal communication $ch.c$, or a time delay d that is a positive real number. We call the events but the time delay *discrete events*, and will use β to range over them. We define the dual of $ch?c$ (denoted by $\overline{ch?c}$) as $ch!c$, and vice versa, and define $comm(ch!c, ch?c)$ or $comm(ch?c, ch!c)$ as the communication $ch.c$. To make our operational semantics more expressive, we will record both the internal events and internal communications that have occurred till now in tr.

The semantics of **skip** and $x := e$ are defined as usual, except that for each, an internal event occurs. Rule (Idle) says that a terminated configuration can keep idle arbitrarily, and then evolves to itself. For input $ch?x$, the input event has to be put in the ready set if it is enabled (In-1); then it may wait for its environment for any time d during which it keeps ready (In-2), or it performs a communication and terminates, with x being assigned and tr extended by the communication, and the ready set being reduced one corresponding to the input (In-3). The semantics of output $ch!e$ is similarly defined by rules (Out-1), (Out-2) and (Out-3). The continuous evolves for d time units if B always holds within this period, during which the ready set is empty (Cont-1), and it terminates at a point when B turns out false at the point or at a right open interval (Cont-2). For communication interrupt, it evolves for d time units if none of the communications io_i is ready (IntP-1), or continues as Q_j if io_j occurs first (IntP-2); or terminates immediately when the continuous terminates (IntP-3). For $P_1\|P_2$, we always assume that the initial states σ_1 and σ_2 are parallelable. There are four rules: both P_1 and P_2 evolve for d time units in case they can delay d time units respectively; or P_1 may progress separately on internal events or external communication events (Par-2), and the symmetric case can be defined similarly (omitted here); or they together perform a synchronized communication (Par-3); or $P_1\|P_2$ terminates when both P_1 and P_2 terminate (Par-4). At last, the semantics for conditional, sequential, internal choice, and repetition is defined as usual.

$$(\mathbf{skip}, \sigma) \xrightarrow{\tau} (\epsilon, \sigma[tr + \tau] \qquad \text{(Skip)}$$

$$(\epsilon, \sigma) \xrightarrow{d} (\epsilon, \sigma[now \mapsto \sigma(now) + d]) \qquad \text{(Idle)}$$

$$(x := e, \sigma) \xrightarrow{\tau} (\epsilon, \sigma[x \mapsto \sigma(e), tr \mapsto \sigma(tr) \cdot \langle \tau, \sigma(now)\rangle]) \qquad \text{(Ass)}$$

$$\frac{\sigma(tr).ch? \notin \sigma(rdy)}{(ch?x, \sigma) \xrightarrow{\tau} (ch?x, \sigma[rdy \mapsto \sigma(rdy) \cup \{\sigma(tr).ch?\}])} \qquad \text{(In-1)}$$

$$\frac{\sigma(tr).ch? \in \sigma(rdy)}{(ch?x, \sigma) \xrightarrow{d} (ch?x, \sigma[now \mapsto \sigma(now) + d], H_{d,i})} \qquad \text{(In-2)}$$

$$\frac{\sigma(tr).ch? \in \sigma(rdy)}{(ch?x,\sigma) \xrightarrow{ch?b} (\epsilon,\ \sigma[x \mapsto b, tr + ch.b, rdy \mapsto \sigma(rdy)\backslash\{\sigma(tr).ch?\}]\,)} \tag{In-3}$$

$$\frac{\sigma(tr).ch! \notin \sigma(rdy)}{(ch!e,\sigma) \xrightarrow{\tau} (ch!e, \sigma[rdy \mapsto \sigma(rdy) \cup \{\sigma(tr).ch!\}])} \tag{Out-1}$$

$$\frac{\sigma(tr).ch! \in \sigma(rdy)}{(ch!e,\sigma) \xrightarrow{d} (ch!e, \sigma[now \mapsto \sigma(now) + d], H_{d,o})} \tag{Out-2}$$

$$\frac{\sigma(tr).ch! \in \sigma(rdy)}{(ch!e,\sigma) \xrightarrow{ch!\sigma(e)} (\epsilon,\ \sigma[tr + ch.\sigma(e), rdy \mapsto \sigma(rdy)\backslash\{\sigma(tr).ch!\}]\,)} \tag{Out-3}$$

$$\frac{\begin{array}{l} S(t) \text{ is a trajectory of } \mathcal{F}(\dot{s},s)=0 \text{ s.t.}(S(0)=\sigma(s) \\ \wedge \forall t \in [0,d].(\mathcal{F}(\dot{S}(t),S(t))=0 \wedge \sigma(B[s \mapsto S(t)]) = true)) \end{array}}{(\langle \mathcal{F}(\dot{s},s)=0\&B\rangle,\sigma) \xrightarrow{d} \begin{pmatrix} \langle \mathcal{F}(\dot{s},s)=0\&B\rangle, \\ \sigma[now \mapsto \sigma(now)+d, s \mapsto S(d)], H_{d,s} \end{pmatrix}} \tag{Cont-1}$$

$$\frac{\begin{array}{l} (\sigma(B) = false) \text{ or } (S(t) \text{ is a trajectory of } \mathcal{F}(\dot{s},s)=0 \text{ s.t.} \\ \exists \varepsilon > 0.(S(0)=\sigma(s) \\ \wedge \forall t \in (0,\varepsilon].(\mathcal{F}(\dot{S}(t),S(t))=0 \wedge \sigma(B[s \mapsto S(t)]) = false))) \end{array}}{(\langle \mathcal{F}(\dot{s},s)=0\&B\rangle,\sigma) \xrightarrow{\tau} (\epsilon, \sigma[s \mapsto \lim_{t\to 0} S(t), tr \mapsto \sigma(tr)\cdot\langle \tau, \sigma(now)\rangle])} \tag{Cont-2}$$

$$\frac{\begin{array}{l} (ch_i*; Q_i, \sigma) \xrightarrow{d} (ch_i*; Q_i, \sigma_i', H_i), \quad \forall i \in I \\ (\langle \mathcal{F}(\dot{s},s)=0\&B\rangle,\sigma) \xrightarrow{d} (\langle \mathcal{F}(\dot{s},s)=0\&B\rangle,\sigma', H) \end{array}}{\begin{array}{l} (\langle \mathcal{F}(\dot{s},s)=0\&B\rangle \trianglerighteq []_{i\in I}(ch_i* \to Q_i),\sigma) \xrightarrow{d} \\ \qquad \begin{pmatrix} \langle \mathcal{F}(\dot{s},s)=0\&B\rangle \trianglerighteq []_{i\in I}(ch_i* \to Q_i), \\ \sigma'[rdy \mapsto \cup_{i\in I}\sigma_i'(rdy)], H[rdy \mapsto \cup_{i\in I}\sigma_i'(rdy)] \end{pmatrix} \end{array}} \tag{IntP-1}$$

$$\frac{(ch_j*; Q_j, \sigma) \xrightarrow{ch_j*} (Q_j, \sigma'), \exists j \in I}{(\langle \mathcal{F}(\dot{s},s)=0\&B\rangle \trianglerighteq []_{i\in I}(ch_i* \to Q_i),\sigma) \xrightarrow{ch_j*} (Q_j,\sigma')} \tag{IntP-2}$$

$$\frac{(\langle \mathcal{F}(\dot{s},s)=0\&B\rangle,\sigma) \xrightarrow{\tau} (\epsilon,\sigma'))}{(\langle \mathcal{F}(\dot{s},s)=0\&B\rangle \trianglerighteq []_{i\in I}(ch_i* \to Q_i),\sigma) \xrightarrow{\tau} (\epsilon,\sigma')} \tag{IntP-3}$$

$$\frac{\begin{array}{l} (P_1,\sigma_1) \xrightarrow{d} (P_1',\sigma_1',H_1), \quad (P_2,\sigma_2) \xrightarrow{d} (P_2',\sigma_2',H_2), \\ \forall ch \in \Sigma(P_1) \cap \Sigma(P_2).\neg((P_1, \sigma_1 \uplus \sigma_2) \xrightarrow{ch*} \wedge (P_2, \sigma_1 \uplus \sigma_2) \xrightarrow{\overline{ch*}}) \end{array}}{(P_1 \parallel P_2, \sigma_1 \uplus \sigma_2) \xrightarrow{d} (P_1' \parallel P_2', (\sigma_1' \uplus \sigma_2'), H_1 \uplus H_2)} \tag{Par-1}$$

$$\frac{(P_1,\sigma_1) \xrightarrow{\beta} (P_1',\sigma_1'), \quad \Sigma(\beta) \notin \Sigma(P_1) \cap \Sigma(P_2)}{(P_1 \parallel P_2, \sigma_1 \uplus \sigma_2) \xrightarrow{\beta} (P_1' \parallel P_2, \sigma_1' \uplus \sigma_2)} \tag{Par-2}$$

$$\frac{(P_1,\sigma_1) \xrightarrow{ch*} (P_1',\sigma_1'), \quad (P_2,\sigma_2) \xrightarrow{\overline{ch*}} (P_2',\sigma_2'),}{(P_1 \parallel P_2, \sigma_1 \uplus \sigma_2) \xrightarrow{comm(ch*,\overline{ch*})} (P_1' \parallel P_2', \sigma_1' \uplus \sigma_2')} \tag{Par-3}$$

$$(\epsilon \parallel \epsilon, \sigma_1 \uplus \sigma_2) \xrightarrow{\tau} (\epsilon, \sigma_1 \uplus \sigma_2) \tag{Par-4}$$

$$\frac{\sigma(B) = \mathit{true}}{(B \to P, \sigma) \xrightarrow{\tau} (P, \sigma[tr + \tau])} \text{ (Cond-1)} \qquad \frac{\sigma(B) = \mathit{false}}{(B \to P, \sigma) \xrightarrow{\tau} (\epsilon, \sigma[tr + \tau])} \text{ (Cond-2)}$$

$$\frac{(P, \sigma) \xrightarrow{\alpha} (P', \sigma', H) \quad P' \neq \epsilon}{(P; Q, \sigma) \xrightarrow{\alpha} (P'; Q, \sigma', H)} \text{ (Seq-1)} \qquad \frac{(P, \sigma) \xrightarrow{\alpha} (\epsilon, \sigma', H)}{(P; Q, \sigma) \xrightarrow{\alpha} (Q, \sigma', H)} \text{ (Seq-2)}$$

$$(P \sqcup Q, \sigma) \xrightarrow{\tau} (P, \sigma[tr + \tau]) \text{ (IntC-1)} \qquad (P \sqcup Q, \sigma) \xrightarrow{\tau} (Q, \sigma[tr + \tau]) \text{ (IntC-2)}$$

$$\frac{(P, \sigma) \xrightarrow{\alpha} (P', \sigma', H) \quad P' \neq \epsilon}{(P^*, \sigma) \xrightarrow{\alpha} (P'; P^*, \sigma', H)} \text{ (Rep-1)} \qquad \frac{(P, \sigma) \xrightarrow{\alpha} (\epsilon, \sigma', H)}{(P^*, \sigma) \xrightarrow{\alpha} (P^*, \sigma', H)} \text{ (Rep-2)}$$

$$(P^*, \sigma) \xrightarrow{\tau} (\epsilon, \sigma[tr + \tau]) \text{ (Rep-3)}$$

where for an internal or communication event β, $\sigma[tr + \beta]$ stands for $\sigma[tr \mapsto \sigma(tr) \cdot \langle \beta, \sigma(now) \rangle]$, and the flow $H_{d,i}$ (or $H_{d,o}$) is defined over time interval $[\sigma(now), \sigma(now) + d]$, such that for any t in the domain, $H_{d,i}(t) = \sigma[now \mapsto t]$ (or $H_{d,o}(t) = \sigma[now \mapsto t]$); and the flow $H_{d,s}$ is defined over time interval $[\sigma(now), \sigma(now) + d]$ such that for any $t \in [\sigma(now), \sigma(now) + d]$, $H_{d,s}(t) = \sigma[now \mapsto t, s \mapsto S(t - \sigma(now))]$, where $S(\cdot)$ is the trajectory as defined in the rule. For any t in the domain, $H_1 \uplus H_2(t) = H_1(t) \uplus H_2(t)$.

Given two flows H_1 and H_2 defined on $[r_1, r_2]$ and $[r_2, r_3]$ respectively, we define the *concatenation* $H_1^\frown H_2$ as the flow defined on $[r_1, r_3]$ such that $H_1^\frown H_2(t)$ is equal to $H_1(t)$ if $t \in [r_1, r_2)$, and $H_2(t)$ if $t \in [r_2, r_3)$. Given a process P and an initial state σ_0, if we have the following sequence of transitions:

$$\begin{array}{l}
(P, \sigma_0) \xrightarrow{\alpha_0} (P_1, \sigma_1, H_1) \\
(P_1, \sigma_1) \xrightarrow{\alpha_1} (P_2, \sigma_2, H_2) \\
\ldots \\
(P_{n-1}, \sigma_{n-1}) \xrightarrow{\alpha_{n-1}} (P_n, \sigma_n, H_n)
\end{array}$$

then we define the sequence $H_1^\frown \ldots ^\frown H_n$ as a *flow* from P_1 to P_n starting from σ_0, and write $(P, \sigma_0) \xrightarrow{\alpha_0 \cdots \alpha_{n-1}} (P_n, \sigma_n, H_1^\frown \ldots ^\frown H_n)$ as an abbreviation of the above transition sequence; and meanwhile, define the sequence $B_1^\frown \ldots ^\frown B_n$ as a *behavior* from P_1 to P_n starting from σ_0, where B_i is H_i if H_i is not empty, empty otherwise if H_i is empty but H_{i+1} is not, σ_i otherwise. Thus, a flow records for each time point the rightmost state, while a behavior records for each time point all the discrete states that occur in execution. Especially, when P_n is ϵ, we will call them *complete flow* and *complete behavior* of P with respect to σ_0 respectively.

6 Hybrid Hoare Logic

HHL was first proposed in [55], which is an extension of Hoare logic to hybrid system, used to specify and reason about hybrid systems modelled by HCSP. The assertion logic of HHL consists of two parts: the first-order logic and Duration Calculus (DC) [97,96]. The former is used to specify discrete events, represented by *pre-* and *post-condition*, while the latter is used to specify continuous evolution. In HHL, a hybrid system is modelled by an HCSP process. So, the proof system of HHL consists of the following three parts: axioms and inference rules for the first-order logic, axioms and inference rules for DC, and axioms and inference rules for the constructs of HCSP. A theorem prover of the logic based on Isabelle/HOL has been implemented, and applied to model and specify Chinese High-Speed Train Control System at Level 3 (CTCS-3) [99].

However, the version of HHL given in [55] can only be used to deal with closed systems, as it lacks compositionality and therefore cannot cope with open systems. Recently, some attempts to define a compositional proof system are undertaken [86,36,93].

Here, we present a revised version of HHL given in[55].

6.1 History Formulas

As indicated before, we will use a subset of DC formulas to record execution history of HCSP processes. The formulas in this subset are denoted as *HF* (*history formula*) and given as follows.

$$HF ::= \ell < T \mid \ell = T \mid \ell > T \mid \lceil S \rceil^0 \mid \neg HF \mid HF_1^\frown HF_2 \mid HF_2 \vee HF_2$$

where ℓ stands for interval length, $T \in \mathbb{R}^+$ is a constant, and S is a state expression, which is a first order formula of $\mathcal{V}(P)$ interpreted as a Boolean function over the time domain, defined by

$$S ::= 1 \mid 0 \mid R(e_1, \ldots, e_n) \mid \neg S \mid S_1 \vee S_2$$

where $R(e_1, \ldots, e_n)$ is a n-ary predicate over expressions $e_1, \ldots, e_n$, normally of the form $p(x_1, \ldots, x_n) \rhd 0$ with $\rhd \in \{\geq, >, =, \neq, \leq, <\}$ and $p(x_1, \ldots, x_n)$ a polynomial in $x_1, \ldots, x_n$.

Informally, the above formulas can be understood as follows:

- $\ell < T$ (resp. $\ell = T$, $\ell > T$) means the length of the reference interval is less than (resp. equal to, greater than) T;
- $\lceil S \rceil^0$ means that the state S is satisfied at the reference point interval, i.e., the considered time point;
- $HF_1^\frown HF_2$ says that the reference interval can be split into two parts such that HF_1 is satisfied on the first segment, while HF_2 holds on the second;
- The logical connectives can be understood in the standard way.

$\lceil S \rceil$ is an abbreviation of $\neg(\mathit{true}^\frown\lceil\neg S\rceil^0{}^\frown\ell > 0)$, which means S holds everywhere on a considered interval, except for its right endpoint. Obviously, we have

$$\begin{array}{ll} \mathit{false} \Leftrightarrow (\ell < 0) & \mathit{true} \Leftrightarrow (\ell = 0) \vee (\ell > 0) \Leftrightarrow \neg(\ell = 0) \vee \lceil S \rceil \\ \lceil S \rceil^\frown\lceil S \rceil \Leftrightarrow \lceil S \rceil & \lceil S \rceil^\frown(\ell = 0) \Leftrightarrow \lceil S \rceil \Leftrightarrow (\ell = 0)^\frown\lceil S \rceil \end{array}$$

In addition, given a history formula *HF*, we use $HF^<$ to denote the internal of *HF*, meaning that *HF* holds on the interval derived from the considered interval by excluding its endpoint. $HF^<$ can be formally defined as follows:

$$\begin{array}{ll} (\ell < T)^< & \overset{\text{def}}{=} (\ell < T) \\ (\ell = T)^< & \overset{\text{def}}{=} (\ell = T) \\ (\ell > T)^< & \overset{\text{def}}{=} \ell > T \\ (\lceil S \rceil^0)^< & \overset{\text{def}}{=} \ell = 0 \\ \lceil S \rceil^< & \overset{\text{def}}{=} \lceil S \rceil \\ (HF_1^\frown HF_2)^< & \overset{\text{def}}{=} (HF_1)^<{}^\frown(HF_2)^< \\ (HF_1 \wedge HF_2)^< & \overset{\text{def}}{=} (HF_1)^< \wedge (HF_2)^< \\ (HF_1 \vee HF_2)^< & \overset{\text{def}}{=} (HF_1)^< \vee (HF_2)^< \end{array}$$

Formally, given a state σ, a state expression S is interpreted as

$$\begin{aligned} \sigma(1) &= 1 \\ \sigma(0) &= 0 \\ \sigma(R(e_1, \ldots, e_n)) &= \begin{cases} 1, & \text{if } R(\sigma(e_1), \ldots, \sigma(e_n)); \\ 0, & \text{otherwise} \end{cases} \\ \sigma(\neg S) &= 1 - \sigma(S) \\ \sigma(S_1 \vee S_2) &= \max\{\sigma(S_1), \sigma(S_2)\} \end{aligned}$$

Thus, given a flow H and a reference interval of the flow $[a, b]$ with $a, b \in \mathrm{Dom}(H)$, and $a \leq b$, we can formally define the meaning of a history formula *HF* inductively as follows:

- $H, [b, e] \models \ell \rhd T$ iff $e - b \rhd T$, where $\rhd \in \{\leq, >, =, \neq, \leq, <\}$;
- $H, [b, e] \models \lceil S \rceil^0$ iff $b = e$, and $H(b)(S) = 1$;
- $H, [b, e] \models \neg HF$ iff $H, [b, e] \not\models HF$;
- $H, [b, e] \models HF_1 \wedge HF_2$ iff $H, [b, e] \models HF_1$ and $H, [b, e] \models HF_2$;
- $H, [b, e] \models HF_1 \vee HF_2$ iff $H, [b, e] \models HF_1$ or $H, [b, e] \models HF_2$;
- $H, [b, e] \models HF_1^\frown HF_2$ iff there is $m \in [b, e]$ such that $H, [b, m] \models HF_1$ and $H, [m, e] \models HF_2$.

6.2 Hoare Assertion

A Hoare assertion of HHL consists of four parts: precondition, process, postcondition and history, written as

$$\{Pre\}P\{Post; HF\}$$

where *Pre* specifies values of $\mathcal{V}(P)$ before an execution of P, *Post* specifies values of $\mathcal{V}(P)$ when P terminates, and *HF* is a formula of $\mathcal{V}(P)$ from the DC subset to describe the execution history of P. HCSP has three kinds of interruptions: boundary interruption like $\langle F(\dot{s}, s) = 0 \wedge B\rangle$, timeout interruption like $\langle F(\dot{s}, s) = 0 \wedge B\rangle \trianglerighteq_d Q$ and communication interruption like $\langle F(\dot{s}, s) = 0 \wedge B\rangle \trianglerighteq \|_{i\in I}(ch_i* \rightarrow Q_i)$. For these three kinds of interruptions, *HF* has to join in reasoning.

Definition 15 (Validity). *We say a Hoare assertion* $\{Pre\}P\{Post; HF\}$ *is* valid*, denoted by* $\models \{Pre\}P\{Post; HF\}$, *iff for any initial state* σ_1, *if* $(P, \sigma_1) \xrightarrow{\alpha^*} (\epsilon, \sigma_2, H)$ *then* $\sigma_1 \models Pre$ *implies* $\sigma_2 \models Post$ *and* $H, [\sigma_1(now), \sigma_2(now)] \models HF$.

For a parallel process, say $P_1 \| ... \| P_n$, the assertion becomes

$$\{Pre_1, ..., Pre_n\}P_1 \| ... \| P_n\{Post_1, ..., Post_n; HF_1, ..., HF_n\}$$

where $Pre_i, Post_i, HF_i$ are (first order or DC) formulas of $\mathcal{V}(P_i)$ $(i = 1, ..., n)$ separately. The validity can be defined similarly.

Another role of *HF* is to specify real-time (continuous) property of an HCSP process, while *Pre* and *Post* can only describe its discrete behaviour. *HF* therefore bridges up the gap between discrete and continuous behaviour of the process. For instance, in Example 8, we may want the plant controller stable after T time units, i.e. after T time units the distance between the trajectory of s and its target s_{targ} must be small. This can be specified through the following assertion.

$$\begin{array}{l}\{s = s_0 \wedge u = u_0 \wedge Ctrl(u_0, s_0), Pre_2\}\ PLC \\ \quad \{Post_1, Post_2; (l = T)^\frown\lceil | s - s_{targ} | \leq \epsilon\rceil, HF_2\}\end{array}$$

where $Ctrl(u, s)$ may express a controllable property, and the other formulas are not elaborated here.

Note that we can essentially put *Pre* and *Post* as parts of history formula *HF* like the form $\lceil Pre\rceil^{0\frown} HF^\frown \lceil Post\rceil^0$. But we did not adopt this way, because separation of specifying and reasoning about discrete behavior and continuous behavior can indeed improve readability and simplify our approach.

6.3 Proof System of HHL

We will omit the axioms and inference rules for the first-order logic and DC, and just concentrate on the axioms and rules for the constructs of HCSP.

1. **Monotonicity**

$$\begin{array}{l}\text{If } \{Pre_1, Pre_2\}P_1 \| P_2\{Post_1, Post_2; HF_1, HF_2\}, \\ \text{and } Pre'_i \Rightarrow Pre_i, Post_i \Rightarrow Post'_i, HF_i \Rightarrow HF'_i (i = 1, 2), \\ \text{then } \{Pre'_1, Pre'_2\}P_1 \| P_2\{Post'_1, Post'_2; HF'_1, HF'_2\}\end{array}$$

   where we use first order logic to reason about $Pre'_i \Rightarrow Pre_i$ and $Post_i \Rightarrow Post'_i$, but use DC to reason about $HF_i \Rightarrow HF'_i$. From now on we will not repeatedly mention this.

2. **Case Analysis**

$$\begin{aligned}&\text{If } \{Pre_{1i}, Pre_2\}P_1 \parallel P_2\{Post_1, Post_2; HF_1, HF_2\}\ (i=1,2),\\ &\text{then } \{Pre_{11} \vee Pre_{12}, Pre_2\}P_1 \parallel P_2\{Post_1, Post_2; HF_1, HF_2\}\end{aligned}$$

Symmetrically,

$$\begin{aligned}&\text{If } \{Pre_1, Pre_{2i}\}P_1 \parallel P_2\{Post_1, Post_2; HF_1, HF_2\}\ (i=1,2),\\ &\text{then } \{Pre_1, Pre_{21} \vee Pre_{22}\}P_1 \parallel P_2\{Post_1, Post_2; HF_1, HF_2\}\end{aligned}$$

3. **Parallel vs Sequential**
These two rules show a simple relation between assertions of a parallel process and its sequential components that can ease a proof.

$$\begin{aligned}&\text{If } \{Pre_1, Pre_2\}P_1 \parallel P_2\{Post_1, Post_2; HF_1, HF_2\}\\ &\text{then } \{Pre_i\}P_i\{Post_i; HF_i\}\ (i=1,2)\end{aligned}$$

and

$$\begin{aligned}&\text{If } \{Pre_i\}P_i\{Post_i; HF_i\}\ (i=1,2),\\ &\text{and } P_i\ (i=1,2) \text{ do not contain communication,}\\ &\text{then } \{Pre_1, Pre_2\}P_1 \parallel P_2\{Post_1, Post_2; HF_1, HF_2\}\end{aligned}$$

4. **Skip**

$$\{Pre\}\mathbf{skip}\{Pre; l=0\},$$

where by $l=0$ we assume that, in comparison with physical device, computation takes no time (i.e. *super dense computation* [60])

5. **Assignment**

$$\{Pre[e/x]\}x := e\{Pre, \lceil x=e \rceil^0\}$$

The precondition and postcondition are copied from Hoare Logic. Here we use $\lceil x=e \rceil^0$ as its history to indicate that x is assigned to e, which takes place at this time point.

6. **Communication**
Since HCSP rejects sharing variables, a communication looks like the output party $(P_1; ch!e)$ assigning to variable x of the input one $(P_2; ch?x)$ a value e. Besides, in order to synchronize both parties, one may have to wait for another. During the waiting of P_i, $Post_i$ must stay true ($i=1$ or 2). We use $const(\mathcal{V}(P))$ to denote $\wedge_{x\in\mathcal{V}(P)}\exists v.\lceil x=v \rceil$, which means that all variables of P keep unchanged except for at the endpoint.

$$\begin{aligned}&\text{If } \{Pre_1, Pre_2\}P_1 \parallel P_2\{Post_1, Post_2; HF_1, HF_2\},\\ &\quad Post_1 \Rightarrow G(e), HF_1 \Rightarrow \ell = c_1, \text{and } HF_2 \Rightarrow \ell = c_2\\ &\text{then } \{Pre_1, Pre_2\}(P_1; ch!e) \parallel (P_2; ch?x)\\ &\quad\{Post_1, G(x) \wedge \exists x.Post_2; HF_1{}^\frown(\lceil Post_1 \rceil \wedge const(\mathcal{V}(P_1)) \wedge \ell = c - c_1),\\ &\quad\ (HF_2{}^\frown(\lceil Post_2 \rceil \wedge const(\mathcal{V}(P_2)) \wedge \ell = c - c_2))^{<\frown}\lceil x=e \rceil^0\}\\ &\text{where } c = \max\{c_1, c_2\}.\end{aligned}$$

Note that for simplicity, in the above rule we just consider a simple case of communication; a rule for the general case of communication

$$(P_1; \talloblong_{i\in I} ch_i* \to Q_{1i}) \parallel (P_2; \talloblong_{j\in J} ch_j* \to Q_{2j}),$$

where $ch_i* = \overline{ch_j*}$ for some $i \in I, j \in J$, can be defined similarly.

Example 9. If

$$\begin{aligned}&\{Pre_1, Pre_2\}P_1 \parallel P_2\\ &\quad\{y = 3, x = 1; (\lceil y = 0\rceil \wedge (l = 3))^\frown \lceil y = 3\rceil^0, \lceil x = 0\rceil \wedge (l = 5)^\frown \lceil x = e\rceil^0\},\end{aligned}$$

we want to deduce through this rule

$$\{Pre_1, Pre_2\}P_1; ch!y \parallel P_2; ch?x\{Post_3, Post_4; HF_3, HF_4\}.$$

Since $(y = 3) \Rightarrow (3 = 3)$, $\lceil y = 0\rceil \wedge (l = 3))^\frown \lceil y = 3\rceil^0 \Rightarrow \ell = 3$, and $(\lceil x = 0\rceil \wedge \ell = 5)^\frown \lceil x = 1\rceil^0 \Rightarrow \ell = 5$, we can conclude that $Post_3$ is $y = 3$, $Post_4$ is $x = 3$, HF_3 is $((\lceil y = 0\rceil \wedge (l = 3))^\frown \lceil y = 3\rceil^{0\frown}(\lceil y = 3\rceil \wedge const(\mathcal{V}(P_1) \cup \{y\}) \wedge \ell = 2)$, and HF_4 is $(\ell = 5^\frown \lceil x = 1\rceil^0)^{<\frown} \lceil x = 3\rceil^0$, which is equivalent to $(\ell = 5^\frown \lceil x = 3\rceil^0$ by the definition of $HF^<$. □

7. **Continuous**
This is about $\langle F(\dot{s}, s) = 0 \wedge B\rangle$, where s can be a vector and F be a group of differential equations, such as

$$\langle(\dot{s}_1 = f_1, ..., \dot{s}_n = f_n) \wedge B\rangle.$$

As indicated in Sec 3, in our framework, we only deal with polynomial differential equations and *semi-algebraic* differential invariants. That is, f_js are polynomials in s_i $(i = 1, ..., n)$, B is a conjunction of polynomial equations and inequalities of s_i $(i = 1, ..., n)$, and differential invariants are also restricted to polynomial equations and inequalities. So, given a polynomial differential invariant *Inv* of $\langle F(\dot{s}, s) = 0 \wedge B\rangle$ with initial values satisfying *Init*, the inference rule for *continuous* can be formulated as follows:

$$\begin{aligned}&\text{If } Init \Rightarrow Inv,\\ &\text{then } \{Init \wedge Pre\}\langle F(\dot{s}, s) = 0 \wedge B\rangle\{Pre \wedge \mathbf{Cl}(Inv) \wedge \mathbf{Cl}(\neg B);\\ &\qquad \lceil Inv \wedge Pre \wedge B\rceil\}\end{aligned}$$

where *Pre* does not contain s, $\mathbf{Cl}(G)$ stands for the *closure* of G[10].
The second rule is about explicit time.

$$\begin{aligned}&\text{If } \{Pre\}\langle F(\dot{s}, s) = 0 \& B\rangle\{Post; HF\}\\ &\text{and } \{Pre \wedge t = 0\}\langle(F(\dot{s}, s) = 0, \dot{t} = 1)\& B\rangle\{t = t_0 \wedge Rg(t_0), HF'\},\\ &\text{then } \{Pre\}\langle F(\dot{s}, s) = 0 \& B\rangle\{Post; HF \wedge Rg(\ell)\}\end{aligned}$$

where t is a clock to count the execution time, and $Rg(t)$ is a constraint on the final value of t which is an arithmetic formula.

[10] When G is constructed by polynomial inequalities through $\wedge$ and $\vee$, $\mathbf{Cl}(G)$ can be obtained from G by replacing $<$ (and $>$) with $\leq$ (and $\geq$) in G.

Example 10. According to the result given in Section 3, it is easy to see that $v \leq v_{ebi}$ is an invariant of $\langle(\dot{s} = v, \dot{v} = a) \wedge v < v_{ebi}\rangle$. Thus, by the continuous rule

$$\{(v = v_0 \leq v_{ebi})\}\langle(\dot{s} = v, \dot{v} = a) \wedge v < v_{ebi}\rangle$$
$$\{(v \leq v_{ebi}) \wedge (v \geq v_{ebi}); \lceil(v \leq v_{ebi}) \wedge (v < v_{ebi})\rceil\}$$

In addition, we can prove that, if the initial values are $v = v_0$ and $t = 0$, and we assume $p \geq a \geq w$, then

$$((v_0 + wt) \leq v \leq (v_0 + pt)) \wedge (v \leq v_{ebi})$$

is an invariant of $\langle(\dot{s} = v, \dot{v} = a, \dot{t} = 1) \wedge v < v_{ebi}\rangle$. So under the assumption $(p \geq a \geq w)$

$$\{(v = v_0 \leq v_{ebi}) \wedge (t = 0)\}\langle(\dot{s} = v, \dot{v} = a, \dot{t} = 1) \wedge v < v_{ebi}\rangle$$
$$\{(v = v_{ebi}) \wedge ((v_0 + wt) \leq v \leq (v_0 + pt)) \wedge \tfrac{v_{ebi}-v_0}{w} \geq t \geq \tfrac{v_{ebi}-v_0}{p};$$
$$\lceil(v < v_{ebi}) \wedge ((v_0 + wt) \leq v \leq (v_0 + pt))\rceil\}$$

Therefore, assuming $(p \geq a \geq w)$ we can have

$$\{(v = v_0 \leq v_{ebi})\}\langle(\dot{s} = v, \dot{v} = a) \wedge v < v_{ebi}\rangle$$
$$\{(v = v_{ebi}); \lceil(v < v_{ebi})\rceil \wedge (\tfrac{v_{ebi}-v_0}{w} \geq l \geq \tfrac{v_{ebi}-v_0}{p})\}$$

$\square$

8. **Sequential.** The rule for sequential composition is very standard, given as follows:
If $\{Pre_1\}P_1\{Post_1; HF_1\}$, and $\{Post_1\}P_2\{Post_2; HF_2\}$
then $\{Pre_1\}P_1; P_2\{Post_2; HF_1^{<}{}^\frown HF_2\}$.
9. **Internal Choice.** The rule for internal choice is standard, given as follows:
If $\{Pre\}P_1\{Post_1; HF_1\}$ and $\{Pre\}P_2\{Post_2; HF_2\}$,
then $\{Pre\}P_1 \sqcup P_2\{Post_1 \vee Post_2; HF_1 \vee HF_2\}$.
10. **Communication Interruption**
There are two rules for communication interruption, the first one says that the continuous part terminates before a communication happens, while the second one states that the continuous evolution is interrupted by a communication.
Rule1: If
   (a) $\{Pre, Pre_R\}\ \langle F(\dot{s}, s) = 0\&B\rangle \parallel R\{Post, Post_R; HF, HF_R\}$,
   (b) for all $i \in I$, $\{Pre, Pre_R\}\ ch_i* \parallel R\{Post_i, Post_R^i; HF_i, HF_R^i\}$,
   (c) $HF \Rightarrow \ell = x$, $\wedge_{i\in I}(HF_i \Rightarrow \ell = x_i) \wedge x < x_i$,
   then
$$\{Pre, Pre_R\}\ \langle F(\dot{s}, s) = 0\&B\rangle \trianglerighteq [\!]_{i\in I}(ch_i* \to Q_i) \parallel R$$
$$\{Post, Post_R; HF, HF_R\}$$
Rule 2: Assume $j \in I$. If
   (a) $\{Pre, Pre_R\}\ \langle F(\dot{s}, s) = 0\&B\rangle \parallel R_1; \overline{ch_j*} \to R_2\{Post, Post_R; HF, HF_R\}$,
   (b) for all $i \in I$, $\{Pre, Pre_R\}\ ch_i* \parallel R_1; \overline{ch_j*}\,\{Post_i, Post_R^i; HF_i, HF_R^i\}$,
   (c) $HF \Rightarrow \ell = x$, $\wedge_{i\in I} HF_i \Rightarrow \ell = x_i$, and $x_j \leq x \wedge \wedge_{i\neq j} x_j \leq x_i$,

(d) $HF \Rightarrow (\ell = x_j \wedge HF_s)^\frown \lceil G(s_0) \rceil^0$,
(e) $\{Post_j \wedge G(s_0), Post_R^j\} Q_j \parallel R_2 \{Post^f, Post_R^f; HF^f, HF_R^f\}$,
then

$$\{Pre, Pre_R\}\ \langle F(\dot{s}, s) = 0 \& B \rangle \trianglerighteq [\!]_{i \in I}(ch_i* \to Q_i) \parallel R_1; \overline{ch_j*} \to R_2$$
$$\{Post^f, Post_R^f; ((HF_s{}^\frown \lceil G(s_0) \rceil^0) \wedge HF_j)^\frown HF^f, HF_R^j{}^\frown HF_R^f\}$$

Note that for simplicity, in Rule 2, we only consider $\langle F(\dot{s}, s) = 0 \& B \rangle \trianglerighteq [\!]_{i \in I}(ch_i* \to Q_i)$ to be parallel with $R_1; \overline{ch_j*}; R_2$. For general case, the rule can be given similarly, without any difficulty.

11. **Repetition**
We can pick up rules from the literature for the repetition. Here we only show a rule which ends off an assertion reasoning.

If $\{Pre_1, Pre_2\} P_1 \parallel P_2 \{Pre_1, Pre_2; HF_1, HF_2\}$,
$HF_i \ \Rightarrow\ (D_i \wedge (l = T))\ (i = 1, 2,\ T > 0)$,
and $D_i^\frown D_i \ \Rightarrow\ D_i$,
then $\{Pre_1, Pre_2\} P_1^* \parallel P_2^* \{Pre_1, Pre_2; \ell = 0 \vee D_1, \ell = 0 \vee D_2\}$

where T is the time consumed by both P_1 and P_2 that can guarantee the synchronisation of the starting point of each repetition.

6.4 Soundness

We only present the case for sequential processes.

Definition 16 (Theorem). *We say a Hoare triple $\{Pre\} P \{Post; HF\}$ is a* theorem*, denoted by $\vdash \{Pre\} P \{Post; HF\}$, iff it is derivable from the above proof system.*

The soundness of the proof system is guaranteed by the following theorem.

Theorem 11 (Soundness). *If $\vdash \{Pre\} P \{Post; HF\}$, then $\models \{Pre\} P \{Post; HF\}$, i.e. every theorem of the proof system is valid.*

7 HHL Prover

In this section, we aim to provide the tool support for verifying whether an HCSP process conforms to its specification written in HHL. Fig. 8 shows the verification architecture of our approach: given an annotated HCSP process in the form of HHL specification, by designing a verification condition generator based on HHL proof system, the specification to be proved is reduced to a set of verification conditions, each of which is either a first-order formula or a DC formula, and the validity of these logical formulas is equivalent to that of the original specification; these logical formulas can then be proved by interactive theorem proving, furthermore, some of which falling in decidable subsets of first-order logic or DC can be proved automatically by designing the corresponding decision procedures.

As shown in Fig. 8, a differential invariant generator is needed for specifying and verifying differential equations. But currently we assume for each differential equation, its invariant is annotated as given, as we have not implemented the results reported in Sec. 3 yet. As one of future work, such an invariant generator will be implemented and integrated.

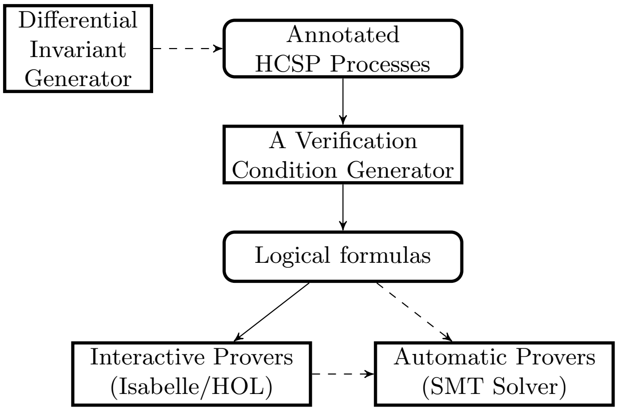

Differential Invariant Generator
Annotated HCSP Processes
A Verification Condition Generator
Logical formulas
Interactive Provers (Isabelle/HOL)
Automatic Provers (SMT Solver)

Fig. 8. Verification Architecture of HCSP Processes

We have mechanized the main part of the verification architecture connected by solid lines shown in Fig. 8 in proof assistant Isabelle/HOL, based on which implemented an interactive theorem prover called *HHL prover* for verifying HHL specifications. The mechanization mainly includes the embedding of HCSP, the assertion languages, i.e., first-order logic (FOL) and DC, and upon them, the embedding of the proof system of HHL in Isabelle/HOL. We adopt the deep embedding approach [14,87] here, which represents the abstract syntax for both HCSP and assertions by new datatypes, and then defines the semantic functions that assign meanings to each construct of the datatypes. It allows us to quantify over the syntactic structures of processes and assertions, and furthermore, make full use of deductive systems for reasoning about assertions written in FOL and DC.

The HHL prover can be downloaded at `https://github.com/iscas/HHL_prover`.

7.1 Expressions

We start from encoding the bottom construct, i.e. expressions, that are represented as a datatype `exp`:

```
datatype exp = RVar string | SVar string | BVar string | Real real
         | String string | Bool bool | exp + exp | exp − exp | exp ∗ exp
```

An expression can be a variable, that can be of three types, `RVar x` for real variable, `SVar x` and `BVar x` for string and boolean variables; a constant, that can be also of the three types, e.g. `Real 1.0`, `String ''CO''` and `Bool True`; an arithmetic expression constructed from operators $+, -, *$. Based on expressions, we can define the assertion languages and the process language HCSP respectively.

7.2 Assertion Language

As we introduced in Sec. 6, there are two assertion logics in HHL: FOL and DC, where the former is used for specifying the pre-/post-conditions and the latter for the execution history of a process respectively. The encodings for both logics consist of two parts: syntax and deductive systems. We will encode the deductive systems in Gentzen's sequent calculus style, which applies backward search to conduct proofs and thus is more widely used in interactive and automated reasoning. A sequent is written as $\Gamma \vdash \Delta$, where both Γ and Δ are sequences of logical formulas, meaning that when all the formulas in Γ are true, then at least one formula in Δ will be true. We will implement a sequent as a truth proposition. The sequent calculus deductive system of a logic is composed of a set of sequent rules, each of which is a relation between a (possibly empty) sequence of sequents and a single sequent. In what follows, we consider to encode FOL and DC respectively.

First-Order Logic. The FOL formulas are constructed from expressions by using relational operators from the very beginning, and can be represented by the following datatype `fform`:

$$\textbf{datatype}\ \texttt{fform} = [\texttt{True}] \mid [\texttt{False}] \mid \texttt{exp}\ [=]\ \texttt{exp} \mid \texttt{exp}\ [<]\ \texttt{exp} \mid [\neg]\ \texttt{fform} \mid \texttt{fform}\ [\vee]\ \texttt{fform} \mid [\forall]\ \textit{string}\ \texttt{fform}$$

The other logical connectives including $[\wedge]$, $[\rightarrow]$, and $[\exists]$ can be derived as normal. For quantified formula $[\forall]$*string* `fform`, the name represented by a string corresponds to a real variable occurring in `fform`. We only consider the quantification over real variables here, but it can be extended to variables of other types (e.g. string and bool) without any essential difficulty. Notice that we add brackets to wrap up the logical constructors in order to avoid the name conflicts between `fform` and the FOL system of Isabelle library. But in sequel, we will remove brackets for readability when there is no confusion in context; and moreover, in order to distinguish between FOL formulas and Isabelle meta-logic formulas, we will use $\Rightarrow$, & and $|$ to represent implication, conjunction and disjunction in Isabelle meta-logic.

Now we need to define the sequent calculus style deductive system for `fform`. The Isabelle library includes an implementation of the sequent calculus of classical FOL with equation, based upon system *LK* that was originally introduced by Gentzen. Our encoding of the sequent calculus for `fform` is built from it directly, but with an extension for dealing with the atomic arithmetic formulas that are defined in `fform`. We define an equivalent relation between the validity of formulas of `fform` and of *bool*, the built-in type of Isabelle logical formulas, represented as follows:

```
formT (f :: fform) ⇔ ⊢ f
```

where the function `formT` transforms a formula of type `fform` to a corresponding formula of *bool*. This approach enables us to prove atomic formulas `f` of `fform` by applying the built-in arithmetic solvers of Isabelle and proving `formT (f)` instead.

Duration Calculus. Encoding DC into different proof assistants has been studied, such as [78] in PVS, and [38,72] in Isabelle/HOL. DC can be considered as an extension of Interval Temporal Logic (ITL) by introducing state durations (here point formulas instead), while ITL an extension of FOL with the introducing of temporal variables and chop modality by regarding intervals instead of points as worlds. Therefore, both [38] and [72] apply an incremental approach to encode ITL on top of an FOL sequent calculus system, and then DC on top of ITL. We will follow a different approach here, to represent DC formulas as a datatype, as a result, the proving of DC formulas can be done by inductive reasoning on the structures of the formulas.

The datatype `dform` encodes the history formulas HF given in Sec. 6:

```
datatype dform = [[True]] | [[False]] | dexp[[=]]dexp | dexp[[<]]dexp
  | [[¬]]dform | dform[[∨]]dform |[[∀]] string dform | pf fform | dform⌢dform
```

We will get rid of double brackets for readability if without confusion in context. The datatype `dexp` defines expressions that are dependent on intervals. As seen from HF, it includes the only temporal variable ℓ for representing the length of the interval, and real constants. Given a state formula `S` of type `fform`, `pf S` encodes the point formula $\lceil S\rceil^0$, and furthermore, the following `high S` encodes formula $\lceil S\rceil$:

```
high :: fform ⇒ dform
high S ≡ ¬ (True ⌢pf (¬S)⌢ ℓ > Real 0)
```

The chop modality $^\frown$ can be encoded as well.

To establish the sequent calculus style deductive system for `dform`, we first define the deductive system for the first-order logic constructors of `dform`, which can be taken directly from the one built for `fform` above, and then define the deductive system related to the new added modalities for DC, i.e. ℓ, $^\frown$ and `pf`.

For ℓ and $^\frown$, we encode the deductive system of ITL from [96], which is presented in Hilbert style. Thus, we need to transform the deductive system to sequent calculus style, and it is not so natural to do. We borrow the idea from [72] that for each modality, define both the left and right introducing rules, e.g., the following implementation

```
LI  : $H, P ⊢ $E ⇒ $H, P⌢(ℓ = Real 0) ⊢ $E
RI  : $H ⊢ P, $E ⇒ $H ⊢ P⌢(ℓ = Real 0), $E
```

where `$H`, `$E` represent arbitrary sequences of logical formulas of type `dform`, encodes the axiom of ITL: $P \leftrightarrow P^\frown(\ell = 0)$. In the same way, for point formula `pf`, we encode the deductive system of DC defined in [96] in sequent calculus style, e.g., the following implementation

```
PFRI  : $H ⊢ (pf S₁⌢pf S₂), $E ⇒ $H ⊢ pf (S₁ ∧ S₂), $E
```

encodes the axiom of DC: $\lceil S_1 \rceil^0 {}^\frown \lceil S_2 \rceil^0 \rightarrow \lceil S_1 \wedge S_2 \rceil^0$.

7.3 HCSP

We represent HCSP processes as a datatype `proc`, and each construct of HCSP can be encoded as a construct in datatype `proc` correspondingly. Most of the encoding is directly a syntactic translation, but with the following exceptions:

– As mentioned in previous sections, in the deductive verification of HCSP process, the role of differential equation is reflected by an differential invariant with respect to the property to be verified, which can be automatically discovered in polynomial cases. So in `proc`, instead of differential equation, we use differential invariant to describe the underlying continuous, and for aiding verification, we also add execution time range of the continuous. Thus, we encode continuous of form $\langle \mathcal{F}(\dot{s}, s) = 0 \& B \rangle$ as `<Inv&B> : Rg`, where `Inv` represents the differential invariant of the continuous, `B` the domain constraint, and `Rg` the range of execution time, of the continuous respectively; and `Inv`, `B` are implemented as formulas of type `fform`, while `Rg` of type `dform`.
– For sequential composition, we encode $P; Q$ as `P; mid; Q`, where `P` and `Q` represent the encodings of P and Q respectively, and `mid` is added to represent the intermediate assertions between P and Q. This is requisite for reducing proof of sequential composition to the ones of its components, and commonly used in theorem proving.
– For parallel composition, we remove the syntax restriction that it can only occur in the outmost scope, thus it is encoded with the same datatype `proc` as other constructs.

7.4 Semantics

In this section, we encode the semantics of HCSP, FOL and DC in Isabelle/HOL. This is done by implementing all the relevant semantic notations and functions defined in Sec. 5 and Sec. 6.

There are two disjoint sets of variables considered in the semantics of HCSP: local variables in $\mathcal{V}(P)$, and system variables $\{now, tr, rdy\}$. Notice that the system variables do not occur in HHL; therefore, we will implement the semantic functions for evaluating them separately. We define type `state` to represent states and each element of it is a function that assigns respective values to (only) process variables. Besides, we define types `now`, `trace`, `ready` to represent system time (i.e. *real*), timed traces and ready sets of communication events. Based on these definitions, we implement the behavior of a process by the following `lbevr` and `sbevr`:

```
type_synonym lbevr = now ⇒(state list)
             sbevr = now ⇒((trace*ready) list)
```

Each local behavior of type `lbevr` associates a sequence of states to each time point, while each system behavior of type `sbevr` associates a sequence of traces and ready sets to each time point. The combination of local behavior and system behavior implements the overall behavior defined in Sec. 5. It should be pointed out that the flow of a process is not implemented explicitly here, but it can always be extracted from the behavior of the process by only keeping the rightmost state in the state list for each time point[11]. Thus, in the following, we always use a behavior whenever a flow is needed.

The expressions of HCSP are interpreted over states. Given a state `s` of type `state` and an expression `e` of type `exp`, the function `evalE(s, e)` defines the value of `e` under the state `s`. Based on the evaluation of expressions, the pre-/post-conditions in the form of `fform`, can be interpreted over states. Given a state `s` of type `state` and a formula `p` of type `fform`, the function `evalF(s, p)` evaluates the truth value of `p` under the state `s`.

As defined in Sec. 6, the history formulas of DC are interpreted over flows and timed intervals. Because the history formulas do not refer to system variables, we interpret them over local behaviors instead of flows. First of all, given a local behaviour `f` of type `lbevr` and a timed interval `[c, d]`, `ievalE(f,`ℓ`, c, d)` returns the value of ℓ, that is `d-c`, under the behavior `f` and the timed interval `[c, d]`. Given a behavior `f` of type `lbevr`, a DC formula `ip`, and a timed interval `[c, d]`, `ievalF(f, ip, c, d)` evaluates the truth value of `ip` under the behavior `f` and the timed interval `[c, d]`. In particular, the point formula and chop can be defined as follows:

```
pf_eval: ievalF (f, pf (P), c, d) = (c=d & evalF (last(f(c)), P))
chop_eval: ievalF (f, P⁀Q, c, d) = ∃k. c<=k & k<=d & ievalF (f, P, c, k)
                                              & ievalF (f, Q, k, d)
```

Thus, `pf(P)` holds, iff the interval is a point interval, and `P` holds at the last state of the state list that is recorded at the time point.

Finally, we implement the operational semantics of HCSP processes. Given a process `P` of type `proc`, a local behavior `f` of type `lbevr`, a system behavior `sf` of type `sbevr`, an event α of type `event`, and a time point `d`, the function `evalP (P, f, sf, d,`α`) = (P', f', sf', d')` represents that, starting to execute from behaviors `f` and `sf`, and time `d`, `P` performs an event α and evolves to `P'` at time `d'`, and produces the new local and system behaviors `f'` and `g'` respectively. It implements exactly the transition relation $(P, \sigma) \xrightarrow{\alpha} (P', \sigma', H)$ defined in Sec. 5, in particular that the initial state σ can be extracted from `f`, `sf`, and `d`, while final state σ' from `f'`, `sf'`, and `d'` respectively.

We explain the semantics of several HCSP constructors as an illustration here. For instance, the transition rule (Ass) of assignment is implemented as follows (only the case for real variables is considered):

[11] Please note the difference between flow and behavior that a flow only records the last state occurring at any time point, while the corresponding behavior records all states occurring at the time point.

```
assignR : evalP (((RVar (x)) :=e), f, sf, d, Tau) =
             (ε, updateVal(f, x, R, e, d), updateTr(sf, Tau, d), d)
```

where `updateVal` adds a new state corresponding to the discrete assignment to the state list recorded at time `d`, and this new state is the same as the initial state except that the value of variable `RVar (x)` is updated by `e`; and `updateTr` adds a `Tau` event to the initial trace and then pushes the resulting trace to the trace list recorded at time `d`. Notice that the termination time is still d, indicating assignment does not take time. As another instance, the transition rule (In-3) of input is implemented as follows:

```
in3 : inList((fst(last(sf(d))), (I ch)), snd(last(sf(d))))
      ⇒ evalP(ch??(RVar(x)), f, sf, d, (Inp ch e)) =
         (ε, updateVal(f, x, R, e, d), removeRdy(
      updateTr(sf, Com(ch, e), d), (fst(last(sf(d))), (I ch)), d), d)
```

where `ch??(RVar(x))` of type `proc` represents an input to real variable. The predicate `inList(...)` represents that the communication event corresponding to input `ch??(RVar(x))`, represented by `(fst(last(sf(d))), (I ch))` of type `ready`, is in the initial ready set (represented by `snd(last(sf(d)))`); and it implements exactly the premise in rule (In-3). It performs an input event `Inp ch e`, and results in the adding of a new state that assigns the value of `e` to `RVar(x)` to the state list at time `d`, and the adding of a new trace increased by the communication `Com(ch,e)` to the trace list at time `d`, and the adding of a new ready set with the removal of the communication event corresponding to input `ch??(RVar(x))` at time `d`. At last, the transition rule (Par-3) for parallel composition is implemented as follows:

```
par3: evalP(P, f, sf, d, (Inp ch e)) = (P', f', sf', d) &
  evalP(Q, f, sf, d, (Outp ch e)) = (Q', f'', sf'', d) ≡
    evalP((P || Q), f, sf, d, a) = ((P' || Q'), f', sf', d)
```

`P` performs an input communication event, while `Q` performs an output communication event along the same channel, as a consequence, a synchronization occurs for `P || Q`. Notice that the resulting behaviors of `P || Q` are exactly the same to those of `P`.

7.5 Proof System of HHL

With the definitions of datatypes `proc`, `fform` and `dform`, it is now easy to encode HHL assertions. First of all, a Hoare assertion for sequential process `P` is implemented as a truth proposition of the form `{Pre} P {Post;HF}`, where `Pre` and `Post` are of type `fform`, and `HF` of type `dform` respectively. A Hoare assertion for parallel process `P||Q` can be implemented in the similar way.

Verification Condition. Based on the inference rules of HHL, we implement the verification condition generator for reasoning about HCSP specifications. The inference rules encoded here are slightly different from those presented in Sec. 6, in the sense that we remove the point formulas for specifying discrete changes in history formulas and use $\ell = 0$ instead. This will not affect the expressiveness and soundness of HHL.

In deep embedding, the effects of assignments are expressed at the level of formulas by substitution. We implement a map as a list of pairs `(exp * exp) list`, and then given a map σ and a formula `p` of type `fform`, we define function `substF(`σ`, p)` to substitute expressions occurring in `p` according to the map σ. Based on this definition, we have the following axiom for assignment `e:=f`:

```
axioms Assignment :
  ⊢ (p → substF ([(e, f)], q)) ∧ (ℓ = Real 0 →G) ⇒ {p} (e :=f) {q; G}
```

According to the rule of assignment, the weakest precondition of `e := f` with respect to postcondition `q` is `substF ([(e, f)], q)`, and on the other hand, the strongest history formula for assignment is `ℓ= Real 0`, indicating that as a discrete action, assignment takes no time. Therefore, `{p} (e :=f) {q;G}` holds, if `p` implies the weakest precondition, and moreover, `G` is implied by the strongest history formula.

For continuous `<Inv & B> : Rg`, we assume that the precondition can be separated into two conjunctive parts: `Init` referring to initial state of continuous variables, and `p` referring to other distinct variables that keep unchanged during continuous evolution. With respect to precondition `Init∧p`, according to the rule of continuous, when it terminates (i.e. `B` is violated), the precondition `p` not relative to initial state, the closures of `Inv` and of `¬B` hold in postcondition; moreover, there are two cases for the history formula: the continuous terminates immediately, represented by `ℓ= Real 0`, or otherwise, throughout the continuous evolution, `p`, `Inv` and `B` hold everywhere except for the endpoint, represented by `high (Inv∧p∧B)`, where both cases satisfy `Rg`.

```
axioms Continuous : ⊢(Init →Inv) ∧ ((p ∧ close(Inv) ∧ close(¬B)) →q)
          ∧ ((((ℓ = Real 0) ∨ (high (Inv ∧ p ∧ B))) ∧ Rg) → G)
          ⇒ {Init ∧ p} <Inv & B> : Rg {q; G}
```

where function `close` returns closure of corresponding formulas. The above axiom says that `{Init∧p} <Inv & B> : Rg {q;G}` holds, if the initial state satisfies invariant `Inv`, and furthermore, both `q` and `G` are implied by the postcondition and the history formula of the continuous with respect to `Init∧p` respectively.

For sequential composition, the intermediate assertions need to be annotated (i.e., `(m, H)` below) to refer to the postcondition and the history formula of the first component. Therefore, the specification `{p} P;(m, H);Q {q;H⌢G}` holds, if both `{p} P {m;H}` and `{m} Q {q;G}` hold, as indicated by the following axiom.

```
axioms Sequence : {p} P {m; H}; {m} Q {q; G} ⇒{p} P; (m, H); Q {q; H⌢G}
```

The following axiom deals with communication `P1; ch!e || P2;ch?x`, where `P1` and `P2` stand for sequential processes. Let `p1` and `p2` be the preconditions for the sequential components respectively, and `(q1, H1)`, `(q2, H2)` the intermediate assertions specifying the postconditions and history formulas for `P1` and `P2` respectively. `r1` and `G1` represent the postcondition and history formula for the left sequential component ended with `ch!e`, while `r2` and `G2` for the right component ended with `ch?x`. `Rg` stands for the execution time range of the whole parallel composition.

```
axioms Communication :
  {p1, p2} P1 || P2 {q1, q2; H1, H2};
  ⊢ (q1 →r1) ∧ (q2 →substF ([(x, e)], r2));
  ⊢ (H1 ⌢ high (q1)) →G1) ∧ (H2 ⌢ high (q2)) →G2);
  ⊢ (((H1 ⌢ high (q1)) ∧ H2) ∨ ((H2 ⌢ high (q2)) ∧ H1)) →Rg;
    ⇒ {p1, p2} ((P1; (q1, H1); ch !! e) || (P2; (q2, H2); ch ?? x))
              {r1, r2; G1 ∧ Rg, G2 ∧ Rg}
```

As shown above, to prove the final specification, the following steps need to be checked: first, the corresponding specification with intermediate assertions as postconditions and history formulas holds for `P1 || P2`; second, after the communication is done, for the sending party, `q1` is preserved, while for the receiving party, `x` is assigned to be `e`. Thus, `r1` must be implied by `q1`, and `q2` implies the weakest precondition of the communicating assignment with respect to `r2`, i.e. `substF ([(x, e)], r2)`; third, for the communication to take place, one party may need to wait for the other party to be ready, in case that `P1` and `P2` do not terminate simultaneously. The left sequential component will result in history formula `H1⌢high (q1)`, in which `high (q1)` indicates that during waiting time, the postcondition of `P1` is preserved, and similarly for the right component. Thus, `G1` and `G2` must be implied by them respectively; and finally, for both cases when one party is waiting for the other, the conjunction of their history formulas must satisfy the execution time `Rg`.

For repetition, we have the following implementation:

```
axioms Repetition :
{p1, p2} P || Q {p1, p2; H1, H2}; ⊢(H1 ⌢ H1 →H1) ∧ (H2 ⌢ H2 →H2)
   ⇒  {p1, p2} P* || Q* {p1, p2; H1 ∨ (ℓ = Real 0), H2 ∨ (ℓ = Real 0)}
```

The above axiom says that the final specification for `P*|| Q*` holds, if the same specification holds for one round of execution, i.e. `P || Q`, and moreover, `H` is idempotent with respect to chop modality. The formula `ℓ= Real 0` indicates that the repetition iterates zero time.

Soundness. To prove the soundness of HHL proof system, we need to have a big step operational semantics for HCSP first, which can be derived directly from the small step semantics given in Sec. 5. Besides, considering that the interpretation of pre-/post-conditions and history formulas are irrelevant to the system behavior and also the events, we will get rid of them in the big step operational semantics, represented by function `evalPB`.

```
axioms
  base: evalP (P, f, sf, d, α) = (ϵ, f', sf', d') ⇔
      evalPB (P, f, d) = (ϵ, f', d')
  ind: evalP (P, f, sf, d, α) = (P', f', sf', d') & evalPB (P', f', d') =
      (ϵ, f'', d'') ⇔ evalPB (P, f, d) = (ϵ, f'', d'')
```

The axioms `base` and `ind` define the cases when `P` terminates after one step transition, and after more than one step transitions respectively.

We can then define the validity of a specification `{p} P {q;H}` with respect to the big operational semantics, as follows:

```
definition Valid :: fform ⇒proc ⇒fform ⇒dform ⇒bool
where Valid (p, P, q, H) = ∀f d f' d'. evalPB (P, f, d) = (ϵ, f', d') ⇒
        evalF (f, p, d) ⇒(evalF (f', q, d') & ievalF (f', H, d, d'))
```

which says that, given a process `P`, for any initial behavior `f` and initial time `d`, if `P` terminates at behavior `f'` and time `d'`, and if the precondition `p` holds under the initial state, i.e. `last(f(d))`, the last state among the state list at initial time, then the postcondition `q` will hold under the final state, i.e. `last(f'(d'))`, the last state among the state list at termination time, and furthermore, the history formula will hold under `f'` between `d` and `d'`.

Based on the above definitions, we have proved the soundness of HHL proof system in Isabelle/HOL, i.e. all the inference rules of the proof system are valid.

8 Case Study: Chinese Train Control System

In this section, we illustrate our approach by modelling and verifying a combined operational scenario of Chinese Train Control System at Level 3 (CTCS-3) with respect to its System Requirement Specification (SRS).

A train at CTCS-3 applies for movement authorities (MAs) from Radio Block Center (RBC) via GSM-Railway (GSM-R) and is guaranteed to move safely in high speed within its MA. CTCS-2 is a backup system of CTCS-3, under which a train applies for MAs from Train Control Center (TCC) via train circuits and balises instead. There are 9 main operating modes in CTCS-3, among which the Full Supervision (FS) and Calling On (CO) modes will be involved in the combined scenario studied in this section. During FS mode, a train needs to know the complete information including its MA, line data, train data and so on; while during CO mode, the on-board equipment of the train cannot confirm explicit routes, thus a train is required to move under constant speed 40km/h.

The operating behavior of CTCS-3 is specified by 14 basic scenarios, all of which cooperate with each other to constitute normal functionality of train control system. The combined scenario considered here integrates the Movement Authority and Level Transition scenarios of CTCS-3, plus a special Mode Transition scenario.

For modeling a scenario, we model each component involved in it as an HCSP process and then combine different parts by parallel composition to form the model of the scenario. In particular, the train participates in each scenario, and the HCSP model corresponding to the train under different scenarios has a very unified structure. Let s be trajectory, v velocity, a acceleration, t clock time of a train respectively. Then we have the following general model for the train:

$$Train \mathrel{\widehat{=}} \begin{pmatrix} \langle \dot{s} = v, \dot{v} = a, \dot{t} = 1 \& \ B\rangle \trianglerighteq []_{i\in I}(io_i \rightarrow P_{comp_i}); \\ Q_{comp} \end{pmatrix}^*$$

where P_{comp_i} and Q_{comp} are discrete computation that takes no time to complete. The train process proceeds as follows: at first the train moves continuously at velocity v and acceleration a; as soon as domain B is violated, or a communication among $\{io_i\}_{i \in I}$ between the train and another component of CTCS-3 takes place, then the train movement is interrupted and shifted to Q_{comp}, or P_{comp_i} respectively; after the discrete computation is done, the train repeats the above process, indicated by * in the model. For each specific scenario, the domain B, communications io_i, and computation P_{comp_i} and Q_{comp} can be instantiated correspondingly. We assume the acceleration a is always in the range $[-b, A]$.

In the rest of this section, we will first model three basic scenarios separately, and then construct a combined scenario from them.

8.1 Movement Authority Scenario

Among all the scenarios, MA is the most basic one and crucial to prohibit trains from colliding with each other. Before moving, the train applies for MA from RBC in CTCS-3 or TCC in CTCS-2, and if it succeeds, it gets the permission to move but only within the MA it owns. An MA is composed of a sequence of segments. Each segment is represented as a tuple $(v_1, v_2, e, mode)$, where v_1 and v_2 represent the speed limits of emergency brake pattern and normal brake pattern by which the train must implement emergency brake and normal brake (thus v_1 is always greater than v_2), e the end point of the segment, and *mode* the operating mode of the train in the segment. We introduce some operations on MAs and segments. Given a non-empty MA α, we define $hd(\alpha)$ to return the first segment of α, and $tl(\alpha)$ the rest sequence after removing the first segment; and given a segment seg, we define $seg.v_1$ to access the element v_1 of seg, and similarly to other elements.

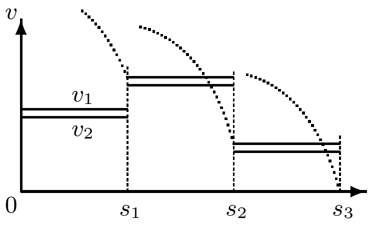


Fig. 9. Static and dynamic speed profiles

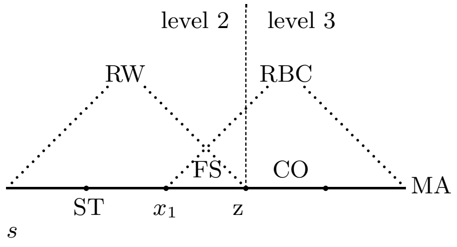


Fig. 10. Level and mode transition

Given an MA, we can calculate its static speed profile and dynamic speed profile respectively. As an illustration, Fig. 9 presents an MA with three segments, separated by points s_1, s_2, and s_3. In the particular case, we assume s_3 the end of the MA, thus the train is required to fully stop at s_3 if the MA is not extended. The static speed profile corresponds to two step functions formed by the two speed limits (i.e. v_1 and v_2) of each segment; and for any segment seg, the dynamic speed profile is calculated down to the higher speed limit of next segment taking into account the train's maximum deceleration (i.e. constant b), and corresponds to the curved function $v^2 + 2b\, s < next(seg).v_1^2 + 2b\, seg.e$, where $next(seg)$ represents the next segment following seg in the considered MA. The train will never be allowed to run beyond the static and dynamic speed profiles.

By specializing the general model of train, we get its specific model in MA scenario. Let B_0 represent the general restriction that the train always moves forward, i.e. $v \geq 0$, or otherwise, the train has already stopped deceleration (denoted by $a \geq 0$). If B_0 fails to hold, the acceleration a needs to be set by a non-negative value in $[0, A]$. Let B_1 denote the case when the speed is less than the lower limit v_2, or otherwise the train has already started to decelerate; and B_2 the case when the speed is less than the higher limit v_1 and not exceeding the dynamic speed profile, or otherwise the train has already started an emergency brake, i.e., the acceleration a is set to be the maximum deceleration b. The above procedure is defined by $Q1_{comp}$ below. For future use, we denote the formula for specifying dynamic speed profile, i.e. $\forall seg : MA\,.\,v^2 + 2b\, s < next(seg).v_1^2 + 2b\, seg.e$, by DSP_Form.

$$\begin{array}{ll}
B_0 & \widehat{=}\ (v \geq 0 \vee a \geq 0 \vee t < Temp + T_{delay}) \\
B_1 & \widehat{=}\ (\forall seg : MA\,.\,v < seg.v_2) \vee a < 0 \vee t < Temp' + T_{delay} \\
B_2 & \widehat{=}\ (\forall seg : MA\,.\,v < seg.v_1 \wedge v^2 + 2b\, s < next(seg).v_1^2 + 2b\, seg.e) \vee a = -b \\
Q1_{comp} & \widehat{=}\ \neg B_0 \rightarrow (Temp := t; \sqcup_{\{0<=c<=A\}} a := c); \\
 & \quad \neg B_1 \rightarrow (Temp' := t; \sqcup_{\{-b<=c<0\}} a := c); \\
 & \quad \neg B_2 \rightarrow a := -b;
\end{array}$$

Notice that we add T_{delay} to clock t to guarantee that the interrupt B_0 can at most occur once every T_{delay} time units, to avoid Zeno behavior. This is in accordance with the real system to check the condition periodically. We adopt this approach several times. In parallel with the train, the RBC or TCC will send MA to the train periodically via communications, and as a consequence, the train will update the MA it owns. We omit the formalization of this process here as it is hardly related to the combined scenario.

8.2 Level Transition

Under CTCS-2, whenever a train passes some specific balises, it can apply for upgrading to CTCS-3 when necessary. It is assumed balises to be equally distributed every δ meters along the track. Let B_3 represent the negation of the case when the train is at level 2 and passing a specific balise. When B_3 is violated, then as specified in $Q2_{comp}$, the following computation will take place: first, the train sends a level upgrade application signal to RBC; as soon as RBC receives the application, it sends back the package (b, x_1, x_2) to the train, where b represents weather RBC approves the application, x_1 the location for starting level upgrade, and x_2 the location for completing level upgrade; if RBC approves the level upgrade (i.e. b is true), the train enters level 2.5 and meanwhile passes the balise. Notice that level 2.5 does not actually exist, but is used only for modelling the middle stage between level 2 and level 3, during which the train will be supervised by both CTCS-2 and CTCS-3. Finally, as soon as the train at level 2.5 reaches location x_2 (the negation denoted by B_4), the level will be set to 3, specified in $Q3_{comp}$. RBC_{lu} defines the behavior of RBC under the level transition scenario.

$$\begin{array}{ll}
B_3 & \mathrel{\widehat{=}} level \neq 2 \ \vee \ s \neq n * \delta \\
B_4 & \mathrel{\widehat{=}} level \neq 2.5 \ \vee \ s \leq LU.x_2 \\
Q2_{comp} & \mathrel{\widehat{=}} \neg B_3 \rightarrow (CH_{LUA}!; CH_{LU}?LU; LU.b \rightarrow level = 2.5; n = n+1); \\
Q3_{comp} & \mathrel{\widehat{=}} \neg B_4 \rightarrow level := 3 \\
RBC_{lu} & \mathrel{\widehat{=}} CH_{LUA}?; \sqcup_{b_{LU} \in \{true, false\}} CH_{LU}!(b, x_1, x_2)
\end{array}$$

8.3 Mode Transition

When a train moves under CTCS-2, it will always check whether its operating mode is equal to the mode of current segment, i.e. $hd(MA).mode$. We denote this condition by B_5, and as soon as it is violated, the train will update its mode to be consistent with *mode* of the segment, specified in $Q4_{comp}$.

$$\begin{array}{ll}
B_5 & \mathrel{\widehat{=}} mode = hd(MA).mode \\
Q4_{comp} & \mathrel{\widehat{=}} \neg B_5 \rightarrow mode := hd(MA).mode
\end{array}$$

We consider the mode transition from Full Supervision (FS) to Calling On (CO) under CTCS-3, which is a little complicated. In the MA application stage, RBC can only grant the train the MAs before the CO segment. The train needs to ask the permission of the driver before moving into a CO segment at level 3. To reflect this specification in modelling, both the speed limits for CO segments are set to be 0. As a consequence, if the train fails to get the permission from the driver, it must stop before the CO segment; but if the train gets the driver's permission, the speed limits of the CO segments will be reset to be positive.

Let B_6 denote the negation of the case when the train is at level 3, and it moves to 300 meters far from the end of current segment, and the mode of next segment is CO. As soon as B_6 is violated, then as specified in $Q5_{comp}$, the following computation will take place: first, the train will report the status to the driver and ask for permission to enter next CO segment via communications; if the driver sends true, the speed limits of next CO segment will be reset to be $40km/h$ and $50km/h$ respectively (abstracted away by function $coma(MA)$). As a consequence, the train is able to enter next CO segment at a positive speed successfully. $Driver_{mc}$ defines the process for the driver under the mode transition scenario.

$$\begin{array}{ll} B_6 & \widehat{=}\ level \neq 3 \vee CO \neq hd(tl(MA)).mode \vee hd(MA).e - s > 300 \\ & \vee t < Temp + T_{delay} \\ Q5_{comp} & \widehat{=}\ CH_{win}!\neg B_6; \neg B_6 \rightarrow Temp := t; CH_{DC}?b_{rConf}; b_{rConf} \rightarrow coma(MA) \\ Driver_{mc} & \widehat{=}\ CH_{win}?b_{win}; b_{win} \rightarrow \sqcup_{b_{sConf} \in \{true, false\}} CH_{DC}!b_{sConf} \end{array}$$

8.4 Combined Scenario and Its Model

We combine the scenarios introduced above, but with the following assumptions for the occurring context:

– The train moves inside an MA it owns, and in the combined scenario, it does not need to apply for new MAs from RBC or TCC;
– There are two adjacent segments in the MA, divided by point z. The train is supervised by CTCS-2 to the left of z and by CTCS-3 to the right, and meanwhile, it is operated by mode FS to the left of z and by mode CO to the right. Thus the locations for mode transition and for level transition are coincident. At the starting point of a CO segment, i.e., location z, both speed limits are initialized to 0 by RBC;
– The train has already got the permission for level transition from RBC which sends $(true, x_1, z)$.

Please see Fig. 10 for an illustration. Based on these assumptions, the train will not cooperate with RBC or TCC temporarily in this combined scenario. Thus, only the train and the driver participate in the combined scenario.

The model of the combined scenario can then be constructed from the models of all the basic scenarios contained in it. The construction takes the following steps: firstly, decompose the process for each basic scenario to a set of sub-processes corresponding to different system components that are involved in the scenario (usually by removing parallel composition on top); secondly, as a component may participate in different basic scenarios, re-construct the process for it based on the sub-processes corresponding to it under these scenarios (usually by conjunction of continuous domain constraints and sequential composition of discrete computation actions); lastly, combine the new obtained processes for all the components via parallel composition. According to this construction process, we get the following HCSP model for the combined scenario:

$$\begin{array}{ll} System & \widehat{=}\ Train^* \parallel Driver^*_{mc} \\ Train & \widehat{=}\ \langle \dot{s} = v, \dot{v} = a, \dot{t} = 1 \&\ B_0 \wedge B_1 \wedge B_2 \wedge B_4 \wedge B_5 \wedge B_6 \rangle; P_{train} \\ P_{train} & \widehat{=}\ Q1_{comp}; Q3_{comp}; Q4_{comp}; Q5_{comp} \end{array}$$

According to SRS of CTCS-3, we hope to prove that the combined scenario satisfies a liveness property, i.e., the train can eventually pass through the location for level transition and mode transition.

8.5 Proof of the Combined Scenario

Under the given assumptions in Section 8.4, we check whether the combined scenario (i.e. model *System*) satisfies a liveness property, i.e., the train will eventually move beyond location z for both level transition and mode transition. In this section, instead of proving the liveness property directly, we provide a machine-checked proof for negation of the livness, which says, after moving for any arbitrary time, the train will always stay before location z. We start from encoding the model *System* and the negation property first.

According to HCSP syntax implemented by `proc`, most encoding of model *System* is a direct translation, except for continuous and sequential composition. Firstly, the continuous of *System* needs to be represented in the form of differential invariants. According to the differential invariant generation method, the differential invariant $(a = -b) \rightarrow DSP\_Form$ is calculated for the continuous, indicating that when the train brakes with maximum deceleration b, it will never exceed the dynamic speed profile. Obviously it is a complement to the domain constraint B_2, saying that the train will never exceed the dynamic speed profile except for the case of emergency brake. We adopt the conjunction of these two formulas, that results in *DSP_Form*, as the final invariant for the continuous. Thus we represent the continuous as `<Inv&B> : Rg`, where `Inv` and `B` correspond to encodings of *DSP_Form* and the domain constraints respectively, and `Rg` is `True`, specifying the executing time of the continuous; Secondly, the intermediate formulas for all sequential composition are added. We finally get the encoding of *System*, represented by `System`, with structure `Train`*`||` `Driver`*.

Now it is turn to encode the negation property, specified by pre/post-conditions, and history formula. The precondition is separated into two parts depending on whether it is relative to initial values, shown by `Init` and `Pre` below:

```
definition Init :: fform where Init ≡(x2 - s > Real 300)
definition Pre :: fform where
Pre ≡ (level = Real 2.5) ∧ (fst (snd (snd (hd (MA)))) = x2)
      ∧ (snd (snd (snd (hd (MA)))) = String ''FS'')
      ∧ (snd (snd (snd (hd (tl (MA))))) = String ''CO'')
      ∧ (fst (hd (MA)) = Real 0) ∧ (fst (snd (hd (MA))) = Real 0)
```

The `Init` represents that the initial position of the train (i.e. `s`) is more than 300 meters away from `x2`. The `Pre` indicates the following aspects: the train moves at level `2.5`, i.e. in process of level transition from CTCS-2 to CTCS-3; the end of current segment is `x2`; the mode of the train in current segment is `''FS''`; the mode of the train in next segment is `''CO''`; and at the end of current segment, both speed limits are initialized to be `0`. Notice that for any segment *seg*, $seg.v_1$ is implemented as `fst (seg)`, and $seg.v_2$ as `fst (snd (seg))`, and so on.

We then get a specification corresponding to the negation property, with the postcondition and history formula for the train to indicate that the train will never pass through location x_2:

```
theorem System : {Init ∧ Pre, True} System {Pre ∧ s <= x2,
                 True; (ℓ = Real 0) ∨ (high (Pre ∧ s <= x2)), True}
```

In Isabelle/HOL, we have proved this specification as a theorem. From this fact, we know that the model `System` for level transition and mode transition fails to conform to the liveness property. This reflects some design flaw for the specifications of related scenarios in CTCS-3.

9 Other Issues: Stability Analysis

In the previous sections, we have discussed the issues of modeling, invariant generation, deductive verification and controller synthesis of hybrid systems. The focus has been on safety properties, that is, properties need to hold at all time. Other important properties of hybrid systems include: *reachability*, which asks whether a given set of target states will be reached in finite time; *stability*, which reflects the influence of small perturbations of initial conditions on the system's trajectories; or *asymptotic stability* which, beyond stability, also cares about the system's convergence behavior when time approaches infinity; and so on. The issues of verification and controller synthesis of hybrid systems for reachability specifications have been investigated in works such as [51,68,29,82,35]. In this section, we will exploit the same techniques developed for invariant generation in Section 3, to automatically generate so-called *relaxed Lyapunov functions* for asymptotic stability analysis of PCDSs. For stability analysis of hybrid systems using tools like *multiple Lyapunov functions*, the readers are referred to such works as [15,16], and the survey papers [27,77] and the citations therein.

9.1 Lyapunov Stability

The following are classic results of stability theory in the sense of Lyapunov. For the details please refer to [50].

Definition 17. *A point $\mathbf{x}_e \in \mathbb{R}^n$ is called an equilibrium point or critical point of a CDS (1) if $\mathbf{f}(\mathbf{x}_e) = \mathbf{0}$.*

It is assumed that $\mathbf{x}_e = \mathbf{0}$ from now on without loss of generality.

Definition 18. *Suppose $\mathbf{0}$ is an equilibrium point of (1). Then*

- $\mathbf{0}$ *is called* **Lyapunov stable** *if for any $\epsilon > 0$, there exists a $\delta > 0$ such that if $\|\mathbf{x}_0\| < \delta$,[12] then the solution $\mathbf{x}(\mathbf{x}_0; t)$ of (1) can be extended to infinity, and $\|\mathbf{x}(\mathbf{x}_0; t)\| < \epsilon$ for all $t \geq 0$.*
- $\mathbf{0}$ *is called* **asymptotically stable** *if it is Lyapunov stable and there exists a $\delta > 0$ such that for any $\|\mathbf{x}_0\| < \delta$, the solution $\mathbf{x}(\mathbf{x}_0; t)$ of (1) satisfies $\lim_{t\to\infty} \mathbf{x}(\mathbf{x}_0; t) = \mathbf{0}$.*

Lyapunov first provided a sufficient condition, using so-called *Lyapunov functions*, for the Lyapunov stability as follows.

[12] For $\mathbf{x} = (x_1, x_2, \ldots, x_n) \in \mathbb{R}^n$, $\|\mathbf{x}\| = \sqrt{\sum_{i=1}^n x_i^2}$ denotes the Euclidean norm of $\mathbf{x}$.

Theorem 12 (Lyapunov Stability Theorem). *Suppose* $\mathbf{0}$ *is an equilibrium point of (1). If there is an open set* $U \subset \mathbb{R}^n$ *with* $\mathbf{0} \in U$*, and a continuously differentiable function* $V : U \to \mathbb{R}$ *such that*

(a) $V(\mathbf{0}) = 0$*,*
(b) $V(\mathbf{x}) > 0$ *for all* $\mathbf{x} \in U \backslash \{\mathbf{0}\}$ *and*
(c) $L^1_{\mathbf{f}} V(\mathbf{x}) \leq 0$ *for all* $\mathbf{x} \in U$*,*

then $\mathbf{0}$ *is a stable equilibrium point. Moreover, if condition (c) is replaced by*

(c)* $L^1_{\mathbf{f}} V(\mathbf{x}) < 0$ *for all* $\mathbf{x} \in U \backslash \{\mathbf{0}\}$*,*

then $\mathbf{0}$ *is an asymptotically stable equilibrium point. Such V satisfying (a), (b) and (c) (or (c*)) is called a* **Lyapunov function***.*

Basically, for asymptotic stability of an equilibrium point of a CDS, Theorem 12 requires a *positive definite* function V with *negative definite* first-order Lie derivative $L^1_{\mathbf{f}} V$ in a neighborhood of the equilibrium. If V has only *negative semi-definite* $L^1_{\mathbf{f}} V$ but no trajectories can stay identically in the zero level set of $L^1_{\mathbf{f}} V$, then the asymptotic stability can also be guaranteed, which is known as the Barbashin-Krasovskii-LaSalle (BKL) Principle.

Theorem 13 (BKL Principle). *Let V be such a function as stated in Theorem 12 with conditions (a), (b) and (c). If the set* $\mathcal{M} \mathrel{\widehat{=}} \{\mathbf{x} \in U \mid L^1_{\mathbf{f}} V(\mathbf{x}) = 0\}$ *does not contain any trajectory of the system other than the trivial trajectory* $\mathbf{x}(t) \equiv \mathbf{0}$*, then* $\mathbf{0}$ *is asymptotically stable.*

9.2 Relaxed Lyapunov Function

Intuitively, a Lyapunov function in Theorem 12 with conditions (a), (b), (c) requires any trajectory starting from $\mathbf{x}_0 \in U$ to stay in the region $\{\mathbf{x} \in \mathbb{R}^n \mid V(\mathbf{x}) \leq V(\mathbf{x}_0)\}$. In the asymptotic stability case, the corresponding V forces any trajectory starting from $\mathbf{x}_0 \in U \backslash \{\mathbf{0}\}$ to transect the boundary $\{\mathbf{x} \in \mathbb{R}^n \mid V(\mathbf{x}) = V(\mathbf{x}_0)\}$, called a *Lyapunov surface*, towards the set $\{\mathbf{x} \in \mathbb{R}^n \mid V(\mathbf{x}) < V(\mathbf{x}_0)\}$. The left picture in Figure 11 illustrates how a Lyapunov function guarantees asymptotic stability.

For any $\mathbf{x}_0 \in U \setminus \{\mathbf{0}\}$, it is not difficult to see that $L^1_{\mathbf{f}} V(\mathbf{x}_0) < 0$ is only a sufficient condition for $\mathbf{x}(\mathbf{x}_0; t)$ to move towards the set $V(\mathbf{x}) < V(\mathbf{x}_0)$. When $L^1_{\mathbf{f}} V(\mathbf{x}_0) = 0$, the transection requirement may still be met if the first non-zero higher order Lie derivative of V at $\mathbf{x}_0$ is negative. In this case, the trajectory may be tangential to a Lyapunov surface at the cross point (see the right picture in Fig. 11). To formalize the above idea, and motivated by the results on invariant generation in Section 3, the following definitions are proposed.

Definition 19 (Pointwise Rank). *Let* $\mathbb{N}^+$ *be the set of positive natural numbers. Given a smooth function* σ *and a smooth vector field* $\mathbf{f}$*, the pointwise rank of* σ *w.r.t.* $\mathbf{f}$ *is defined as the function* $\nu_{\sigma,\mathbf{f}} : \mathbb{R}^n \to \mathbb{N}^+ \cup \{\infty\}$ *given by*

$$\nu_{\sigma,\mathbf{f}}(\mathbf{x}) = \begin{cases} \infty, & \text{if } \forall k \in \mathbb{N}^+ .\, L^k_{\mathbf{f}} \sigma(\mathbf{x}) = 0, \\ \min\{k \in \mathbb{N}^+ \mid L^k_{\mathbf{f}} \sigma(\mathbf{x}) \neq 0\}, & \text{otherwise.} \end{cases}$$

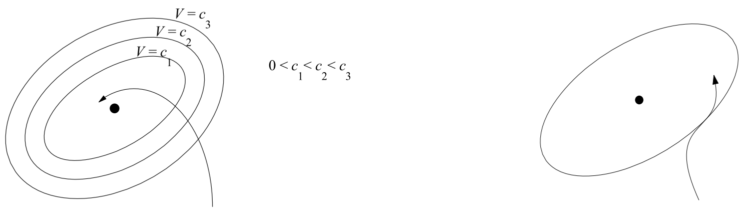


Fig. 11. Trajectories transecting Lyapunov surfaces

Example 11. For $\mathbf{f} = (-x, y)$ and $p(x, y) = x + y^2$, by Example 1, we have $\nu_{p,\mathbf{f}}(0, 0) = \infty$, $\nu_{p,\mathbf{f}}(1, 1) = 1$, $\nu_{p,\mathbf{f}}(2, 1) = 2$.

Actually, $\nu_{\sigma,\mathbf{f}}$ is almost the same as the pointwise rank function $\gamma_{p,\mathbf{f}}$ defined in Section 3.2. The only difference is that for $\nu_{\sigma,\mathbf{f}}$, the zeroth order Lie derivative is not considered.

Definition 20 (Transverse Set). *Given a smooth function σ and a smooth vector field $\mathbf{f}$, the* transverse set *of σ w.r.t $\mathbf{f}$ is defined as*

$$\mathrm{Trans}_{\sigma,\mathbf{f}} \mathrel{\widehat{=}} \{\mathbf{x} \in \mathbb{R}^n \mid \nu_{\sigma,\mathbf{f}}(\mathbf{x}) < \infty \wedge L_{\mathbf{f}}^{\nu_{\sigma,\mathbf{f}}(\mathbf{x})} \sigma(\mathbf{x}) < 0\} \ .$$

Actually, $\mathrm{Trans}_{\sigma,\mathbf{f}}$ is defined in the same manner as the transverse set $Trans_{\mathbf{f}\uparrow p}$ in Definition 14, using a different definition of pointwise rank function.

Using transverse set, condition (c*) in Theorem 12 can be relaxed to give a new criterion for asymptotic stability.

Theorem 14. *Suppose $\mathbf{0}$ is an equilibrium point of (1) with smooth vector field $\mathbf{f}$. If there is an open set $U \subset \mathbb{R}^n$ with $\mathbf{0} \in U$, and a smooth function $V : U \to \mathbb{R}$ such that*

(a) $V(\mathbf{0}) = 0$,
(b) $V(\mathbf{x}) > 0$ for all $\mathbf{x} \in U \backslash \{\mathbf{0}\}$ and
(c) $\mathbf{x} \in \mathrm{Trans}_{V,\mathbf{f}}$ for all $\mathbf{x} \in U \backslash \{\mathbf{0}\}$,

then $\mathbf{0}$ is an asymptotically stable equilibrium.

Proof. First notice that condition (c) implies $L_{\mathbf{f}}^1 V(\mathbf{x}) \leq 0$ for all $\mathbf{x} \in U \backslash \{\mathbf{0}\}$. Then according to Theorem 13, in order to show the asymptotic stability of $\mathbf{0}$, it is sufficient to show that $\mathcal{M} \mathrel{\widehat{=}} \{\mathbf{x} \in U \mid L_{\mathbf{f}}^1 V(\mathbf{x}) = 0\}$ contains no nontrivial trajectory of (1).

If not, let $\mathbf{x}(t)$, $t \geq 0$ be such a trajectory contained in $\mathcal{M}$ other than $\mathbf{x}(t) \equiv \mathbf{0}$. Then for all $t \geq 0$, $L_{\mathbf{f}}^1 V(\mathbf{x}(t)) = 0$ and $\mathbf{x}(t) \neq \mathbf{0}$. By (c), $\mathbf{x}_0 \mathrel{\widehat{=}} \mathbf{x}(0) \in \mathrm{Trans}_{V,\mathbf{f}}$. Then by Definition 20, we get the *Taylor Formula* of $L_{\mathbf{f}}^1 V(\mathbf{x}(t))$ at $t = 0$:

$$\begin{aligned} L^1_{\mathbf{f}}V(\mathbf{x}(t)) &= L^1_{\mathbf{f}}V(\mathbf{x}_0) + L^2_{\mathbf{f}}V(\mathbf{x}_0)\cdot t + \cdots \\ &\qquad + L_{\mathbf{f}}^{\nu_{V,\mathbf{f}}(\mathbf{x}_0)}V(\mathbf{x}_0)\cdot \frac{t^{\nu_{V,\mathbf{f}}(\mathbf{x}_0)-1}}{(\nu_{V,\mathbf{f}}(\mathbf{x}_0)-1)!} + o(t^{\nu_{V,\mathbf{f}}(\mathbf{x}_0)-1}) \\ &= L_{\mathbf{f}}^{\nu_{V,\mathbf{f}}(\mathbf{x}_0)}V(\mathbf{x}_0)\cdot \frac{t^{\nu_{V,\mathbf{f}}(\mathbf{x}_0)-1}}{(\nu_{V,\mathbf{f}}(\mathbf{x}_0)-1)!} + o(t^{\nu_{V,\mathbf{f}}(\mathbf{x}_0)-1}) \,. \end{aligned} \tag{19}$$

Since $L_{\mathbf{f}}^{\nu_{V,\mathbf{f}}(\mathbf{x}_0)}V(\mathbf{x}_0) < 0$, the formula (19) shows that there exists an $\epsilon > 0$ s.t. $\forall t \in (0,\epsilon).\, L^1_{\mathbf{f}}V(\mathbf{x}(t)) < 0$, which contradicts the fact $\forall t \geq 0.\, L^1_{\mathbf{f}}V(\mathbf{x}(t)) = 0$. □

Definition 21 (Relaxed Lyapunov Function). *We refer to the function V in Theorem 14 as a relaxed Lyapunov function, denoted by RLF for short.*

9.3 Automatically Discovering Polynomial RLFs for PCDSs

Given a PCDS, the process of automatically discovering polynomial RLFs is as follows:

I. A parametric polynomial $p(\mathbf{u},\mathbf{x})$ (also called a template) is predefined as a candidate for RLF;
II. The conditions for $p(\mathbf{u},\mathbf{x})$ to be an RLF, i.e. (a), (b) and (c) in Theorem 14, are encoded into a first-order polynomial formula φ;
III. Constraint ϕ on the parameters $\mathbf{u}$ is obtained by applying QE to φ, and any instantiation of $\mathbf{u}$ from ϕ yields an RLF $p_{\mathbf{u}_0}(\mathbf{x})$.

Step II in the above process, i.e. encoding of the three conditions in Theorem 14, is crucial to automatic RLF generation. In particular, we have to show that for any polynomial $p(\mathbf{x})$ and PVF $\mathbf{f}$, the transverse set $\mathrm{Trans}_{p,\mathbf{f}}$ can be represented by first-order polynomial formulas. In fact, all the results established for $Trans_{\mathbf{f}\uparrow p}$ in Section 3.3 apply to $\mathrm{Trans}_{p,\mathbf{f}}$ here.

Theorem 15 (Fixed Point Theorem). *Given a polynomial p and a PVF $\mathbf{f}$, if $L^{i+1}_{\mathbf{f}}p \in \langle L^1_{\mathbf{f}}p, \cdots, L^i_{\mathbf{f}}p\rangle$, then for all $m > i$, $L^m_{\mathbf{f}}p \in \langle L^1_{\mathbf{f}}p, \cdots, L^i_{\mathbf{f}}p\rangle$.*

Theorem 16 (Rank Theorem). *Given a polynomial p and a PVF $\mathbf{f}$, for any $\mathbf{x} \in \mathbb{R}^n$, if $\nu_{p,\mathbf{f}}(\mathbf{x}) < \infty$ then $\nu_{p,\mathbf{f}}(\mathbf{x}) \leq N_{p,\mathbf{f}}$, where*

$$N_{p,\mathbf{f}} \mathrel{\widehat{=}} \min\{i \in \mathbb{N}^+ \mid L^{i+1}_{\mathbf{f}}p(\mathbf{x}) \in \langle L^1_{\mathbf{f}}p(\mathbf{x}), \cdots, L^i_{\mathbf{f}}p(\mathbf{x})\rangle\} \,.$$

Theorem 17 (Parametric Rank Theorem). *Given a parametric polynomial $p \mathrel{\widehat{=}} p(\mathbf{u},\mathbf{x})$ and a PVF $\mathbf{f}$, for all $\mathbf{x} \in \mathbb{R}^n$ and all $\mathbf{u}_0 \in \mathbb{R}^w$, $\nu_{p_{\mathbf{u}_0},\mathbf{f}}(\mathbf{x}) < \infty$ implies $\nu_{p_{\mathbf{u}_0},\mathbf{f}}(\mathbf{x}) \leq N_{p,\mathbf{f}}$, where*

$$N_{p,\mathbf{f}} \mathrel{\widehat{=}} \min\{i \in \mathbb{N}^+ \mid L^{i+1}_{\mathbf{f}}p(\mathbf{u},\mathbf{x}) \in \langle L^1_{\mathbf{f}}p(\mathbf{u},\mathbf{x}), \cdots, L^i_{\mathbf{f}}p(\mathbf{u},\mathbf{x})\rangle\} \,. \tag{20}$$

Theorem 18. *Given a parametric polynomial* $p \widehat{=} p(\mathbf{u}, \mathbf{x})$ *and a PVF* $\mathbf{f}$*, for any* $\mathbf{u}_0 \in \mathbb{R}^w$ *and any* $\mathbf{x} \in \mathbb{R}^n$*,* $\mathbf{x} \in \mathrm{Trans}_{p_{\mathbf{u}_0}, \mathbf{f}}$ *if and only if* $\mathbf{u}_0$ *and* $\mathbf{x}$ *satisfy* $\varphi_{p,\mathbf{f}}$*, where*

$$\varphi_{p,\mathbf{f}} \widehat{=} \bigvee_{1 \le i \le N_{p,\mathbf{f}}} \varphi^i_{p,\mathbf{f}} \quad \textit{with} \tag{21}$$

$$\varphi^i_{p,\mathbf{f}} \widehat{=} \Big(\bigwedge_{1 \le j \le i-1} L^j_{\mathbf{f}} p(\mathbf{u}, \mathbf{x}) = 0 \Big) \wedge L^i_{\mathbf{f}} p(\mathbf{u}, \mathbf{x}) < 0$$

and $N_{p,\mathbf{f}}$ *defined in (20).*

All the proofs of the above theorems can be given in exactly the same way as in Section 3.3. The details are omitted here and can be found in [57].

Now the main result on automatically generating polynomial RLFs for PCDSs can be stated as the following theorem.

Theorem 19 (Main Result). *Given a PCDS* $\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x})$ *with* $\mathbf{f}(\mathbf{0}) = \mathbf{0}$*, a parametric polynomial* $p \widehat{=} p(\mathbf{u}, \mathbf{x})$*, and* $\mathbf{u}_0 = (u_{1_0}, u_{2_0}, \ldots, u_{w_0}) \in \mathbb{R}^w$*, then* $p_{\mathbf{u}_0}$ *is an RLF of the PCDS if and only if there exists* $r_0 \in \mathbb{R}, r_0 > 0$*, such that* $(u_{1_0}, u_{2_0}, \ldots, u_{w_0}, r_0)$ *satisfies* $\phi_{p,\mathbf{f}} \widehat{=} \phi^1_{p,\mathbf{f}} \wedge \phi^2_{p,\mathbf{f}} \wedge \phi^3_{p,\mathbf{f}}$*, where*

$$\phi^1_{p,\mathbf{f}} \widehat{=} p(\mathbf{u}, \mathbf{0}) = 0\,, \tag{22}$$

$$\phi^2_{p,\mathbf{f}} \widehat{=} \forall \mathbf{x}.(\|\mathbf{x}\|^2 > 0 \wedge \|\mathbf{x}\|^2 < r^2 \rightarrow p(\mathbf{u}, \mathbf{x}) > 0)\,, \tag{23}$$

$$\phi^3_{p,\mathbf{f}} \widehat{=} \forall \mathbf{x}.(\|\mathbf{x}\|^2 > 0 \wedge \|\mathbf{x}\|^2 < r^2 \rightarrow \varphi_{p,\mathbf{f}}) \tag{24}$$

with $\varphi_{p,\mathbf{f}}$ *defined in (21).*

Proof. First, in Theorem 14, the existence of an open set U is equivalent to the existence of an open ball $\mathcal{B}(\mathbf{0}, r_0) \widehat{=} \{\mathbf{x} \in \mathbb{R}^n \mid \|\mathbf{x}\| < r_0\}$. Then according to Theorem 18, it is easy to check that (22), (23) and (24) are direct translations of conditions (a), (b) and (c) in Theorem 14. □

According to Theorem 19, we can follow the three steps at the beginning of Section 9.3 to discover polynomial RLFs for PCDSs. This method is *relatively complete* because we can discover all possible polynomial RLFs in the form of a predefined template, and thus can find all polynomial RLFs by enumerating all polynomial templates for a given PCDS.

9.4 Simplification and Implementation

When constructing $\phi_{p,\mathbf{f}}$ in Theorem 19, computation of $N_{p,\mathbf{f}}$ is a time-consuming work. Furthermore, when $N_{p,\mathbf{f}}$ is a large number the resulting $\phi_{p,\mathbf{f}}$ could be a huge formula, for which QE is infeasible in practice. Regarding this, in the following the complexity of RLF generation is reduced in two aspects:

1) some of the QE problems arising in RLF generation can be reduced to so-called *real root classification* (RRC for short) problems, which can be solved in a more efficient way than standard QE problems;

2) RLF can be searched for in a stepwise manner: if an RLF can be obtained by solving constraints involving only lower order Lie derivatives, there is no need to resort to higher order ones.

The following three lemmas are needed to explain the first aspect.

Lemma 5. *Suppose* $\mathbf{f}$ *is a smooth vector field,* σ *is a smooth function defined on an open set* $U \subseteq \mathbb{R}^n$*, and* $L^1_{\mathbf{f}}\sigma(\mathbf{x}) \leq 0$ *for all* $\mathbf{x} \in U$*. Then for any* $\mathbf{x} \in U$*,* $\nu_{\sigma,\mathbf{f}}(\mathbf{x}) < \infty$ *implies* $\mathbf{x} \in \mathrm{Trans}_{\sigma,\mathbf{f}}$ *.*

Proof. Suppose there is an $\mathbf{x}_0 \in U$ such that $\nu_{\sigma,\mathbf{f}}(\mathbf{x}_0) < \infty$ and $L_{\mathbf{f}}^{\nu_{\sigma,\mathbf{f}}(\mathbf{x}_0)}\sigma(\mathbf{x}_0) > 0$. Let $\mathbf{x}(t)$ be the trajectory of $\mathbf{f}$ starting from $\mathbf{x}_0$. Then from

$$\begin{aligned} L^1_{\mathbf{f}}\sigma(\mathbf{x}(t)) &= L^1_{\mathbf{f}}\sigma(\mathbf{x}_0) + L^2_{\mathbf{f}}\sigma(\mathbf{x}_0)\cdot t + \cdots \\ &\qquad + L_{\mathbf{f}}^{\nu_{\sigma,\mathbf{f}}(\mathbf{x}_0)}\sigma(\mathbf{x}_0)\cdot \frac{t^{\nu_{\sigma,\mathbf{f}}(\mathbf{x}_0)-1}}{(\nu_{\sigma,\mathbf{f}}(\mathbf{x}_0)-1)!} + o(t^{\nu_{\sigma,\mathbf{f}}(\mathbf{x}_0)}-1) \\ &= L_{\mathbf{f}}^{\nu_{\sigma,\mathbf{f}}(\mathbf{x}_0)}\sigma(\mathbf{x}_0)\cdot \frac{t^{\nu_{\sigma,\mathbf{f}}(\mathbf{x}_0)-1}}{(\nu_{\sigma,\mathbf{f}}(\mathbf{x}_0)-1)!} + o(t^{\nu_{\sigma,\mathbf{f}}(\mathbf{x}_0)}-1) \end{aligned} \tag{25}$$

we can see that there exists an $\epsilon > 0$ such that $\forall t \in (0,\epsilon).\, L^1_{\mathbf{f}}\sigma(\mathbf{x}(t)) > 0$, which contradicts $L^1_{\mathbf{f}}\sigma(\mathbf{x}) \leq 0$ for all $\mathbf{x} \in U$. □

Lemma 6. *Suppose* $\mathbf{f}$ *is a smooth vector field,* σ *is a smooth function defined on an open set* $U \subseteq \mathbb{R}^n$*, and* $L^1_{\mathbf{f}}\sigma(\mathbf{x}) \leq 0$ *for all* $\mathbf{x} \in U$*. Then for any* $\mathbf{x} \in U$*,* $\nu_{\sigma,\mathbf{f}}(\mathbf{x}) < \infty$ *implies* $\nu_{\sigma,\mathbf{f}}(\mathbf{x}) = 2k+1$ *for some* $k \in \mathbb{N}$ *.*

Proof. If there is an $\mathbf{x}_0 \in U$ such that $\nu_{\sigma,\mathbf{f}}(\mathbf{x}_0) < \infty$ and $\nu_{\sigma,\mathbf{f}}(\mathbf{x}_0) = 2k$ for some $k \in \mathbb{N}^+$, then by Lemma 5 we have $L_{\mathbf{f}}^{\nu_{\sigma,\mathbf{f}}(\mathbf{x}_0)}\sigma(\mathbf{x}_0) < 0$. Then by (25) we can see there exists an $\epsilon > 0$ such that $\forall t \in (-\epsilon, 0).\, L^1_{\mathbf{f}}\sigma(\mathbf{x}(t)) > 0$, which contradicts $L^1_{\mathbf{f}}\sigma(\mathbf{x}) \leq 0$ for all $\mathbf{x} \in U$. □

Lemma 7. *Suppose* $\mathbf{f}$ *is a PVF and* $p(\mathbf{x})$ *is a polynomial, and* $L^1_{\mathbf{f}}p(\mathbf{x}) \leq 0$ *for all* $\mathbf{x}$ *in an open set* $U \subseteq \mathbb{R}^n$*. Then for any* $\mathbf{x} \in U$*,* $\mathbf{x} \in \mathrm{Trans}_{p,\mathbf{f}}$ *if and only if* $\mathbf{x}$ *is not a common root of the sequence of polynomials*

$$L^1_{\mathbf{f}}p(\mathbf{x}),\, L^3_{\mathbf{f}}p(\mathbf{x}),\, \ldots,\, L_{\mathbf{f}}^{(2K_0+1)}p(\mathbf{x})\ ,$$

where $K_0 \widehat{=} \lfloor \frac{N_{p,\mathbf{f}}-1}{2} \rfloor$[13] *and* $N_{p,\mathbf{f}}$ *is defined in Theorem 16.*

Proof. $(\Rightarrow)$ Actually K_0 has been chosen is such a way that $2K_0+1$ is the largest odd number less than or equal to $N_{p,\mathbf{f}}$, i.e. $2K_0+1 = N_{p,\mathbf{f}}$ or $2K_0+1 = N_{p,\mathbf{f}}-1$. Suppose $\mathbf{x}_0 \in \mathrm{Trans}_{p,\mathbf{f}}$ and $L^1_{\mathbf{f}}p(\mathbf{x}_0) = L^3_{\mathbf{f}}p(\mathbf{x}_0) = \cdots = L_{\mathbf{f}}^{(2K_0+1)}p(\mathbf{x}_0) = 0$. From Lemma 6 we know that $\nu_{p,\mathbf{f}}(\mathbf{x}_0)$ is an odd number. Thus $\nu_{p,\mathbf{f}}(\mathbf{x}_0) \geq 2K_0+1+2 > N_{p,\mathbf{f}}$, which contradicts Theorem 16 .

$(\Leftarrow)$ If $\mathbf{x}_0$ is not a common root of $L^1_{\mathbf{f}}p(\mathbf{x}),\, L^3_{\mathbf{f}}p(\mathbf{x}),\, \ldots,\, L_{\mathbf{f}}^{(2K_0+1)}p(\mathbf{x})$, then $\nu_{p,\mathbf{f}}(\mathbf{x}_0) < \infty$. By Lemma 5 we get $\mathbf{x}_0 \in \mathrm{Trans}_{p,\mathbf{f}}$. □

[13] For $0 \leq r \in \mathbb{R}$, we have $\lfloor r \rfloor \in \mathbb{N}$ and $r - 1 < \lfloor r \rfloor \leq r$.

Now a simplified version of Theorem 19 can be given as follows.

Theorem 20. *Given a PCDS* $\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x})$ *with* $\mathbf{f}(\mathbf{0}) = \mathbf{0}$*, a parametric polynomial* $p \mathrel{\widehat{=}} p(\mathbf{u}, \mathbf{x})$*, and* $\mathbf{u}_0 = (u_{1_0}, u_{2_0}, \dots, u_{w_0}) \in \mathbb{R}^w$*, then* $p_{\mathbf{u}_0}$ *is an RLF of the PCDS if and only if there exists* $r_0 \in \mathbb{R}, r_0 > 0$ *such that* $(u_{1_0}, u_{2_0}, \dots, u_{w_0}, r_0)$ *satisfies* $\psi_{p,\mathbf{f}} \mathrel{\widehat{=}} \psi^1_{p,\mathbf{f}} \wedge \psi^2_{p,\mathbf{f}} \wedge \psi^3_{p,\mathbf{f}} \wedge \psi^4_{p,\mathbf{f}}$ *, where*

$$\psi^1_{p,\mathbf{f}} \mathrel{\widehat{=}} p(\mathbf{u}, \mathbf{0}) = 0\,, \tag{26}$$

$$\psi^2_{p,\mathbf{f}} \mathrel{\widehat{=}} \forall \mathbf{x}.(\|\mathbf{x}\|^2 > 0 \;\wedge\; \|\mathbf{x}\|^2 < r^2 \to p(\mathbf{u}, \mathbf{x}) > 0)\,, \tag{27}$$

$$\psi^3_{p,\mathbf{f}} \mathrel{\widehat{=}} \forall \mathbf{x}.(\|\mathbf{x}\|^2 < r^2 \to L^1_{\mathbf{f}} p(\mathbf{u}, \mathbf{x}) \le 0)\,, \tag{28}$$

$$\psi^4_{p,\mathbf{f}} \mathrel{\widehat{=}} \forall \mathbf{x}.(0 < \|\mathbf{x}\|^2 < r^2 \to L^1_{\mathbf{f}} p(\mathbf{x}) \ne 0 \vee L^3_{\mathbf{f}} p(\mathbf{x}) \ne 0 \vee \cdots \vee L^{(2K_0+1)}_{\mathbf{f}} p(\mathbf{x}) \ne 0) \tag{29}$$

with K_0 *defined in Lemma 7.*

Proof. By combining Theorem 14 with Lemma 7 we can get the results immediately. □

In Theorem 20, constraints (26), (27) and (28) have relatively small sizes and can be solved by QE tools, while (29) can be handled more efficiently as an RRC problem of parametric semi-algebraic systems.

Definition 22. *A parametric semi-algebraic system (PSAS for short) is a conjunction of polynomial formulas of the following form:*

$$\begin{cases} p_1(\mathbf{u}, \mathbf{x}) = 0, ..., p_r(\mathbf{u}, \mathbf{x}) = 0, \\ g_1(\mathbf{u}, \mathbf{x}) \ge 0, ..., g_k(\mathbf{u}, \mathbf{x}) \ge 0, \\ g_{k+1}(\mathbf{u}, \mathbf{x}) > 0, ..., g_l(\mathbf{u}, \mathbf{x}) > 0, \\ h_1(\mathbf{u}, \mathbf{x}) \ne 0, ..., h_m(\mathbf{u}, \mathbf{x}) \ne 0, \end{cases} \tag{30}$$

where $r \ge 1, l \ge k \ge 0, m \ge 0$ *and all* p_i*'s,* g_i*'s and* h_i*'s are in* $\mathbb{Q}[\mathbf{u}, \mathbf{x}] \setminus \mathbb{Q}$.

For a PSAS, the interesting problem is so-called *real root classification*, that is, to determine conditions on the parameters $\mathbf{u}$ such that the given PSAS has certain prescribed number of distinct real solutions. Theories on real root classification of PSASs were developed in [90,91]. A computer algebra tool named DISCOVERER [89] was developed to implement these theories.

Given a PSAS $\mathcal{P}$ with n indeterminates and s polynomial equations, it was argued in [19] that CAD-based QE on $\mathcal{P}$ has complexity doubly exponential in n. In contrast, the RRC approach has complexity singly exponential in n and doubly exponential in t, where t is the dimension of the ideal generated by the s polynomials. Therefore RRC can dramatically reduce the complexity of RRC problems especially when t is much less than n.

For RLF generation, to solve (29) we can define a PSAS

$$\mathcal{P} \mathrel{\widehat{=}} \begin{cases} L^1_{\mathbf{f}} p(\mathbf{u}, \mathbf{x}) = 0,\ L^3_{\mathbf{f}} p(\mathbf{u}, \mathbf{x}) = 0,\ \dots,\ L^{(2K_0+1)}_{\mathbf{f}} p(\mathbf{u}, \mathbf{x}) = 0 \\ -\|\mathbf{x}\|^2 > -r^2, \|\mathbf{x}\|^2 > 0 \end{cases} .$$

Then the command *RealRootClassification*$(\mathcal{P}, 0)$ in DISCOVERER returns conditions on $\mathbf{u}$ and r such that $\mathcal{P}$ has NO solutions. In practice, $\mathcal{P}$ can be constructed in a stepwise manner. That is, $L_{\mathbf{f}}^{(2i+1)}p(\mathbf{u}, \mathbf{x}) = 0$ for $0 \leq i \leq K_0$ can be added to $\mathcal{P}$ one by one. Based on the above ideas, an RLF generation algorithm (Algorithm 1) is proposed to implement Theorem 20.

Algorithm 1. Relaxed Lyapunov Function Generation

```
1  Input: $\mathbf{f} \in \mathbb{Q}^n[x_1, \ldots, x_n]$ with $\mathbf{f}(\mathbf{0}) = \mathbf{0}$, $p \in \mathbb{Q}[u_1, \ldots, u_w, x_1, \ldots, x_n]$
2  Output: $Res \subseteq \mathbb{R}^{w+1}$
3  $i := 1$; $Res := \emptyset$; $L_{\mathbf{f}}^1 p := (\nabla p, \mathbf{f})$;
4  $\mathcal{P} := \|\mathbf{x}\|^2 > 0 \wedge -\|\mathbf{x}\|^2 > -r^2$;
5  $Res^0 := \mathrm{QE}(\psi_{p,\mathbf{f}}^1 \wedge \psi_{p,\mathbf{f}}^2 \wedge \psi_{p,\mathbf{f}}^3)$;
6  if $Res^0 = \emptyset$ then
7      return $\emptyset$;
8  else
9      repeat
10         $\mathcal{P} := \mathcal{P} \wedge L_{\mathbf{f}}^i p = 0$;
11         $Res := Res^0 \cap \mathrm{RRC}(\mathcal{P}, 0)$;
12         if $Res \neq \emptyset$ then
13             return $Res$;
14         else
15             $L_{\mathbf{f}}^{i+1} p := (\nabla L_{\mathbf{f}}^i p, \mathbf{f})$;
16             $L_{\mathbf{f}}^{i+2} p := (\nabla L_{\mathbf{f}}^{i+1} p, \mathbf{f})$;
17             $i := i + 2$;
18     until $L_{\mathbf{f}}^i p \in \langle L_{\mathbf{f}}^1 p, L_{\mathbf{f}}^2 p, \ldots, L_{\mathbf{f}}^{i-1} p \rangle$;
19 return $\emptyset$;
```

Remark 1. In Algorithm 1,

- $\psi_{p,\mathbf{f}}^1$ $\psi_{p,\mathbf{f}}^2$ and $\psi_{p,\mathbf{f}}^3$ in Line 5 are defined in (26), (27) and (28) respectively;
- QE in line 5 is done in a computer algebra tool like Redlog [30] or QEPCAD [17];
- RRC in line 11 stands for the *RealRootClassification* command in DISCOVERER;
- in Line 18 the loop test can be done by the *IdealMembership* command in Maple™ [61].

Termination of Algorithm 1 is guaranteed by Theorem 15 and Theorem 17; correctness of Algorithm 1 is guaranteed by Theorem 20.

9.5 Example

We illustrate the method for RLF generation using the following example.

Example 12. Consider the PCDS

$$\begin{pmatrix} \dot{x} \\ \dot{y} \end{pmatrix} = \begin{pmatrix} -x + y^2 \\ -xy \end{pmatrix} \tag{31}$$

with a unique equilibrium point $O(0,0)$. We want to establish the asymptotic stability of O.

First, the linearization of (31) at O has the coefficient matrix

$$A = \begin{pmatrix} -1 & 0 \\ 0 & 0 \end{pmatrix}$$

with eigenvalues -1 and 0, so none of the principles of stability for linear systems can be applied. Besides, a homogeneous quadratic Lyapunov function $x^2 + axy + by^2$ for verifying asymptotic stability of (31) does not exist in $\mathbb{R}^2$, because

$$\forall x \forall y. \begin{pmatrix} x^2 + y^2 > 0 \to \big(x^2 + axy + by^2 > 0 \\ \wedge\, 2x\dot{x} + ay\dot{x} + ax\dot{y} + 2by\dot{y} < 0\big) \end{pmatrix}$$

is *false*. However, if we try to find an RLF in $\mathbb{R}^2$ for (31) using the simple template $p \widehat{=} x^2 + ay^2$ with a the indeterminate, then Algorithm 1 returns $a = 1$. This means (31) has an RLF $x^2 + y^2$, and O is asymptotically stable. See Fig. 12 for an illustration.

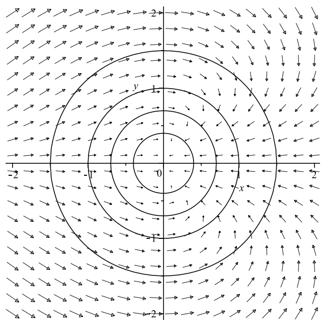


Fig. 12. Vector field and Lyapunov surfaces in Example 12

From this example, we can see that RLFs really extend the class of functions that can be used for asymptotic stability analysis, and the method for automatically discovering polynomial RLFs can save the effort in finding conventional Lyapunov functions in some cases.

10 Conclusion

In this tutorial, we have developed a theoretical and practical foundation for deductive verification of hybrid systems, which includes a selection of topics related to modeling, analysis, and logic of hybrid systems. We choose HCSP as the formal modeling language for hybrid systems, due to its more compositionality and scalability compared to automata-based approach. In order to guarantee the correct functioning of hybrid systems, we have defined a specification logic, called HHL, for specifying and reasoning about the behavior of HCSP, both discrete and continuous, based on first-order logic and DC respectively. However, the logic is not compositional, thus fails to manage more complex HCSP models. The compositionality of the specification logic is one main topic we are working on now.

The specification logic for HCSP uses differential invariants for proving correctness about differential equations instead of their solutions, because solutions of differential equations may not even be expressible. To support this, we have invented a relative complete method for generating polynomial invariants for polynomial differential equations, based on higher-order Lie derivatives and the theory of polynomial ideal.

As a complement of logic-based verification, synthesis provides another approach to ensuring hybrid systems meet given requirements. It focuses on designing a controller for a system such that under the controller, the system is guaranteed to satisfy the given requirement. Based on the differential invariant generation method, we have solved the switching controller synthesis problem with respect to a safety requirement in the context of hybrid automata; and on the other hand, we have also studied the switching controller synthesis problem with respect to an optimality requirement by reducing it to a constraint solving problem.

For tool support, we have implemented a theorem prover for verifying HCSP models in Isabelle/HOL, called HHL prover, which takes an annotated HCSP model in the form of HHL specification as input, and by interactive theorem proving, checks whether the model conforms to the annotated property. The automated verification is not considered yet, and both verification techniques and their implementation to support this will be one of our future work.

Finally, we have demonstrated that our logic-based verification techniques can be used successfully for verifying safety and liveness properties in practical train control systems. In particular, we considered a combined scenario originating from the Chinese High-Speed Train Control System at Level 3 (CTCS-3), and reached a verification result in HHL prover indicating a design error of the combined scenario in CTCS-3. We will consider how to apply our approach to more case studies, among which one direction will be on the safety checking of the other scenarios of CTCS-3 and their all possible combinations.

Acknowledgements. First of all, we thank all the collaborators of the joint work presented in this tutorial for their great contribution. The deductive approach to formal verification of hybrid systems based on Hybrid Hoare Logic

(HHL) was pioneered by Prof. Chaochen Zhou, to whom the basic principles of HHL should be mainly attributed; Prof. Dimitar P. Guelev contributes to the joint work of developing improved versions of HHL; formal modelling and verification of Chinese High-Speed Train Control System (CTCS-3), in particular the scenario reported in Sec. 8, is the result of our long-term collaboration with a research team led by Prof. Tao Tang in Beijing Jiaotong University; Dr. Jidong Lv, Mr. Zhao Quan and Mr. Liang Zou are involved intensively in modelling and verifying CTCS-3; Mr. Liang Zou also helps with the development of HHL prover; Dr. Jiang Liu is one of the major contributors to the work on invariant generation and stability analysis of hybrid systems; the part on (optimal) controller synthesis of hybrid systems is joint work with Prof. Deepak Kapur and Prof. Kim G. Larsen.

We also Thank Prof. Lu Yang, Prof. Bican Xia, Prof. Shaofa Yang, Dr. Ming Xu, Dr. Jiaqi Zhu, Yang Gao, Danqing Guo and many other colleagues for their valuable comments and helpful discussions on the topics of this tutorial.

The work in this tutorial has been supported mainly by projects NSFC-91118007, NSFC-6110006, and National Science and Technology Major Project of China (Grant No. 2012ZX01039-004).

References

1. Alur, R., Courcoubetis, C., Halbwachs, N., Henzinger, T., Ho, P.H., Nicollin, X., Olivero, A., Sifakis, J., Yovine, S.: The algorithmic analysis of hybrid systems. Theoretical Computer Science 138(1), 3–34 (1995)
2. Alur, R., Dang, T., Esposito, J., Hur, Y., Ivančić, F., Kumar, V., Mishra, P., Pappas, G., Sokolsky, O.: Hierarchical modeling and analysis of embedded systems. Proceedings of the IEEE 91(1), 11–28 (2003)
3. Alur, R., Henzinger, T., Ho, P.H.: Automatic symbolic verification of embedded systems. IEEE Transactions on Software Engineering 22(3), 181–201 (1996)
4. Alur, R.: Formal verification of hybrid systems. In: EMSOFT 2011, pp. 273–278. ACM, New York (2011)
5. Alur, R., Courcoubetis, C., Henzinger, T.A., Ho, P.H.: Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In: Grossman, R.L., Nerode, A., Ravn, A.P., Rischel, H. (eds.) HS 1991 and HS 1992. LNCS, vol. 736, pp. 209–229. Springer, Heidelberg (1993)
6. Alur, R., Dang, T., Ivančić, F.: Counterexample-guided predicate abstraction of hybrid systems. Theor. Comput. Sci. 354(2), 250–271 (2006)
7. Alur, R., Dang, T., Ivančić, F.: Predicate abstraction for reachability analysis of hybrid systems. ACM Trans. Embed. Comput. Syst. 5(1), 152–199 (2006)
8. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science 126(2), 183–235 (1994)
9. Alur, R., Henzinger, T.A.: Modularity for timed and hybrid systems. In: Mazurkiewicz, A., Winkowski, J. (eds.) CONCUR 1997. LNCS, vol. 1243, pp. 74–88. Springer, Heidelberg (1997)
10. Asarin, E., Bournez, O., Dang, T., Maler, O., Pnueli, A.: Effective synthesis of switching controllers for linear systems. Proceedings of the IEEE 88(7), 1011–1025 (2000)

11. Asarin, E., Bournez, O., Dang, T., Maler, O.: Approximate reachability analysis of piecewise-linear dynamical systems. In: Lynch, N.A., Krogh, B.H. (eds.) HSCC 2000. LNCS, vol. 1790, pp. 20–31. Springer, Heidelberg (2000)
12. Audemard, G., Bozzano, M., Cimatti, A., Sebastiani, R.: Verifying industrial hybrid systems with MathSAT. Electronic Notes in Theoretical Computer Science 119(2), 17–32 (2005)
13. Bensalem, S., Bozga, M., Fernández, J.-C., Ghirvu, L., Lakhnech, Y.: A transformational approach for generating non-linear invariants. In: Palsberg, J. (ed.) SAS 2000. LNCS, vol. 1824, pp. 58–72. Springer, Heidelberg (2000)
14. Boulton, R.J., Gordon, A., Gordon, M.J.C., Harrison, J., Herbert, J., Tassel, J.V.: Experience with embedding hardware description languages in HOL. In: Proceedings of the IFIP TC10/WG 10.2 International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience, pp. 129–156. North-Holland Publishing Co. (1992)
15. Branicky, M.: Stability of switched and hybrid systems. In: CDC 1994, vol. 4, pp. 3498–3503 (1994)
16. Branicky, M.: Multiple Lyapunov functions and other analysis tools for switched and hybrid systems. IEEE Transactions on Automatic Control 43(4), 475–482 (1998)
17. Brown, C.W.: QEPCAD B: A program for computing with semi-algebraic sets using CADs. SIGSAM Bull. 37, 97–108 (2003)
18. Cassez, F., Jessen, J.J., Larsen, K.G., Raskin, J.-F., Reynier, P.-A.: Automatic synthesis of robust and optimal controllers – an industrial case study. In: Majumdar, R., Tabuada, P. (eds.) HSCC 2009. LNCS, vol. 5469, pp. 90–104. Springer, Heidelberg (2009)
19. Chen, Y., Xia, B., Yang, L., Zhan, N.: Generating polynomial invariants with DISCOVERER and QEPCAD. In: Jones, C.B., Liu, Z., Woodcock, J. (eds.) Formal Methods and Hybrid Real-Time Systems. LNCS, vol. 4700, pp. 67–82. Springer, Heidelberg (2007)
20. Chutinan, A., Krogh, B.H.: Verification of polyhedral-invariant hybrid automata using polygonal flow pipe approximations. In: Vaandrager, F.W., van Schuppen, J.H. (eds.) HSCC 1999. LNCS, vol. 1569, pp. 76–90. Springer, Heidelberg (1999)
21. Clarke, E., Fehnker, A., Han, Z., Krogh, B., Stursberg, O., Theobald, M.: Verification of hybrid systems based on counterexample-guided abstraction refinement. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 192–207. Springer, Heidelberg (2003)
22. Clarke, E., Emerson, E.: Design and synthesis of synchronization skeletons using branching time temporal logic. In: Kozen, D. (ed.) Logic of Programs. LNCS, vol. 131, pp. 52–71. Springer, Heidelberg (1982)
23. Colón, M.A., Sankaranarayanan, S., Sipma, H.B.: Linear invariant generation using non-linear constraint solving. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 420–432. Springer, Heidelberg (2003)
24. Cox, D., Little, J., O'Shea, D.: Ideals, Varieties, and Algorithms: An Introduction to Computational Algebraic Geometry and Commutative Algebra, 2nd edn. Springer (1997)
25. Damm, W., Pinto, G., Ratschan, S.: Guaranteed termination in the verification of LTL properties of non-linear robust discrete time hybrid systems. In: Peled, D.A., Tsay, Y.-K. (eds.) ATVA 2005. LNCS, vol. 3707, pp. 99–113. Springer, Heidelberg (2005)

26. de Moura, L., Bjørner, N.S.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
27. DeCarlo, R., Branicky, M., Pettersson, S., Lennartson, B.: Perspectives and results on the stability and stabilizability of hybrid systems. Proceedings of the IEEE 88(7), 1069–1082 (2000)
28. Deshpande, A., Göllü, A., Varaiya, P.: SHIFT: A formalism and a programming language for dynamic networks of hybrid automata. In: Antsaklis, P., Kohn, W., Nerode, A., Sastry, S. (eds.) HS 1996. LNCS, vol. 1273, pp. 113–133. Springer, Heidelberg (1997)
29. Ding, J., Tomlin, C.: Robust reach-avoid controller synthesis for switched nonlinear systems. In: CDC 2010, pp. 6481–6486 (2010)
30. Dolzmann, A., Seidl, A., Sturm, T.: Redlog User Manual, Edition 3.1, for Redlog Version 3.06 (Reduce 3.8) edn. (2006)
31. Eker, J., Janneck, J., Lee, E.A., Liu, J., Liu, X., Ludvig, J., Sachs, S., Xiong, Y., Neuendorffer, S.: Taming heterogeneity — the Ptolemy approach. Proceedings of the IEEE 91(1), 127–144 (2003)
32. Floyd, R.W.: Assigning Meanings to Programs. In: Schwartz, J.T. (ed.) Proceedings of a Symposium on Applied Mathematics, vol. 19, pp. 19–31 (1967)
33. Fränzle, M., Hahn, E.M., Hermanns, H., Wolovick, N., Zhang, L.: Measurability and safety verification for stochastic hybrid systems. In: HSCC 2011, pp. 43–52. ACM, New York (2011)
34. Fränzle, M., Teige, T., Eggers, A.: Engineering constraint solvers for automatic analysis of probabilistic hybrid automata. The Journal of Logic and Algebraic Programming 79(7), 436–466 (2010)
35. Girard, A.: Controller synthesis for safety and reachability via approximate bisimulation. CoRR abs/1010.4672 (2010), `http://arxiv.org/abs/1010.4672`
36. Guelev, D., Wang, S., Zhan, N.: Hoare reasoning about HCSP in the duration calculus (submitted, 2013)
37. He, J.: From CSP to hybrid systems. In: A Classical Mind: Essays in Honour of C. A. R. Hoare, pp. 171–189. Prentice Hall International (UK) Ltd., Hertfordshire (1994)
38. Heilmann, S.T.: Proof Support for Duration Calculus. Ph.D. thesis, Technical University of Denmark (1999)
39. Henzinger, T.: The theory of hybrid automata. In: LICS 1996, pp. 278–292 (July 1996)
40. Henzinger, T.A., Ho, P.H.: Algorithmic analysis of nonlinear hybrid systems. In: Wolper, P. (ed.) CAV 1995. LNCS, vol. 939, pp. 225–238. Springer, Heidelberg (1995)
41. Henzinger, T.A., Kopke, P.W., Puri, A., Varaiya, P.: What's decidable about hybrid automata? In: STOC 1995, pp. 373–382. ACM, New York (1995)
42. Henzinger, T.A., Sifakis, J.: The embedded systems design challenge. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 1–15. Springer, Heidelberg (2006)
43. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM 12(10), 576–580 (1969)
44. Jha, S., Seshia, S.A., Tiwari, A.: Synthesis of optimal switching logic for hybrid systems. In: EMSOFT 2011, pp. 107–116. ACM, New York (2011)
45. Julius, A., Girard, A., Pappas, G.: Approximate bisimulation for a class of stochastic hybrid systems. In: American Control Conference 2006, pp. 4724–4729 (2006)

46. Julius, A., Pappas, G.: Probabilistic testing for stochastic hybrid systems. In: CDC 2008, pp. 4030–4035 (2008)
47. Kapur, D., Shyamasundar, R.K.: Synthesizing controllers for hybrid systems. In: Maler, O. (ed.) HART 1997. LNCS, vol. 1201, pp. 361–375. Springer, Heidelberg (1997)
48. Kapur, D.: Automatically generating loop invariants using quantifier elimination. In: Baader, F., Baumgartner, P., Nieuwenhuis, R., Voronkov, A. (eds.) Deduction and Applications (2005)
49. Kapur, D., Zhan, N., Zhao, H.: Synthesizing switching controllers for hybrid systems by continuous invariant generation. CoRR abs/1304.0825 (2013), `http://arxiv.org/abs/1304.0825`
50. Khalil, H.K.: Nonlinear Systems, 3rd edn. Prentice Hall (December 2001)
51. Koo, T.J., Pappas, G.J., Sastry, S.S.: Mode switching synthesis for reachability specifications. In: Di Benedetto, M.D., Sangiovanni-Vincentelli, A.L. (eds.) HSCC 2001. LNCS, vol. 2034, pp. 333–346. Springer, Heidelberg (2001)
52. Krantz, S., Parks, H.: A Primer of Real Analytic Functions, 2nd edn. Birkhäuser, Boston (2002)
53. Lafferriere, G., Pappas, G.J., Yovine, S.: Symbolic reachability computation for families of linear vector fields. Journal of Symbolic Computation 32(3), 231–253 (2001)
54. Liu, J., Zhan, N., Zhao, H.: Computing semi-algebraic invariants for polynomial dynamical systems. ArXiv e-prints (Febraury 2011), `http://arxiv.org/abs/1102.0705`
55. Liu, J., Lv, J., Quan, Z., Zhan, N., Zhao, H., Zhou, C., Zou, L.: A calculus for hybrid CSP. In: Ueda, K. (ed.) APLAS 2010. LNCS, vol. 6461, pp. 1–15. Springer, Heidelberg (2010)
56. Liu, J., Zhan, N., Zhao, H.: Computing semi-algebraic invariants for polynomial dynamical systems. In: EMSOFT 2011, pp. 97–106. ACM, New York (2011)
57. Liu, J., Zhan, N., Zhao, H.: Automatically discovering relaxed Lyapunov functions for polynomial dynamical systems. Mathematics in Computer Science 6(4), 395–408 (2012)
58. Lynch, N., Segala, R., Vaandrager, F., Weinberg, H.: Hybrid I/O automata. In: Alur, R., Sontag, E.D., Henzinger, T.A. (eds.) HS 1995. LNCS, vol. 1066, pp. 496–510. Springer, Heidelberg (1996)
59. Maler, O., Manna, Z., Pnueli, A.: From timed to hybrid systems. In: Huizing, C., de Bakker, J.W., Rozenberg, G., de Roever, W.-P. (eds.) REX 1991. LNCS, vol. 600, pp. 447–484. Springer, Heidelberg (1992)
60. Manna, Z., Pnueli, A.: Verifying hybrid systems. In: Grossman, R.L., Ravn, A.P., Rischel, H., Nerode, A. (eds.) HS 1991 and HS 1992. LNCS, vol. 736, pp. 4–35. Springer, Heidelberg (1993)
61. Maplesoft: Maple 14 User Manual, `http://www.maplesoft.com/documentation_center/`
62. Naur, P.: Proof of algorithms by general snapshots. BIT Numerical Mathematics 6(4), 310–316 (1966)
63. Nicollin, X., Olivero, A., Sifakis, J., Yovine, S.: An approach to the description and analysis of hybrid systems. In: Grossman, R.L., Ravn, A.P., Rischel, H., Nerode, A. (eds.) HS 1991 and HS 1992. LNCS, vol. 736, pp. 149–178. Springer, Heidelberg (1993)
64. Platzer, A.: Differential-algebraic dynamic logic for differential-algebraic programs. J. Log. and Comput. 20(1), 309–352 (2010)

65. Platzer, A., Clarke, E.M.: Computing differential invariants of hybrid systems as fixedpoints. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 176–189. Springer, Heidelberg (2008)
66. Platzer, A., Clarke, E.M.: Formal verification of curved flight collision avoidance maneuvers: A case study. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 547–562. Springer, Heidelberg (2009)
67. Prajna, S., Jadbabaie, A., Pappas, G.: A framework for worst-case and stochastic safety verification using barrier certificates. IEEE Transactions on Automatic Control 52(8), 1415–1428 (2007)
68. Prajna, S.: Optimization-based methods for nonlinear and hybrid systems verification. Ph.D. thesis, California Institute of Technology (January 2005)
69. Prajna, S., Jadbabaie, A.: Safety verification of hybrid systems using barrier certificates. In: Alur, R., Pappas, G.J. (eds.) HSCC 2004. LNCS, vol. 2993, pp. 477–492. Springer, Heidelberg (2004)
70. Puri, A., Varaiya, P.: Decidability of hybrid systems with rectangular differential inclusions. In: Dill, D.L. (ed.) CAV 1994. LNCS, vol. 818, pp. 95–104. Springer, Heidelberg (1994)
71. Queille, J., Sifakis, J.: Specification and verification of concurrent systems in CESAR. In: Dezani-Ciancaglini, M., Montanari, U. (eds.) Programming 1982. LNCS, vol. 137, pp. 337–351. Springer, Heidelberg (1982)
72. Rasmussen, T.M.: Interval Logic — Proof Theory and Theorem Proving. Ph.D. thesis, Technical University of Denmark (2002)
73. Ratschan, S., She, Z.: Safety verification of hybrid systems by constraint propagation based abstraction refinement. In: Morari, M., Thiele, L. (eds.) HSCC 2005. LNCS, vol. 3414, pp. 573–589. Springer, Heidelberg (2005)
74. Sankaranarayanan, S.: Automatic invariant generation for hybrid systems using ideal fixed points. In: HSCC 2010, pp. 221–230. ACM, New York (2010)
75. Sankaranarayanan, S., Sipma, H.B., Manna, Z.: Constructing invariants for hybrid systems. In: Alur, R., Pappas, G.J. (eds.) HSCC 2004. LNCS, vol. 2993, pp. 539–554. Springer, Heidelberg (2004)
76. Sankaranarayanan, S., Sipma, H.B., Manna, Z.: Non-linear loop invariant generation using Gröbner bases. In: POPL 2004, pp. 318–329. ACM, New York (2004)
77. Shorten, R., Wirth, F., Mason, O., Wulff, K., King, C.: Stability criteria for switched and hybrid systems. SIAM Rev. 49(4), 545–592 (2007)
78. Skakkebaek, J.U., Shankar, N.: Towards a duration calculus proof assistant in PVS. In: Langmaack, H., de Roever, W.-P., Vytopil, J. (eds.) FTRTFT 1994. LNCS, vol. 863, pp. 660–679. Springer, Heidelberg (1994)
79. Taly, A., Gulwani, S., Tiwari, A.: Synthesizing switching logic using constraint solving. In: Jones, N.D., Müller-Olm, M. (eds.) VMCAI 2009. LNCS, vol. 5403, pp. 305–319. Springer, Heidelberg (2009)
80. Taly, A., Gulwani, S., Tiwari, A.: Synthesizing switching logic using constraint solving. International Journal on Software Tools for Technology Transfer 13(6), 519–535 (2011)
81. Taly, A., Tiwari, A.: Deductive verification of continuous dynamical systems. In: Kannan, R., Kumar, K.N. (eds.) FSTTCS 2009. LIPIcs, vol. 4, pp. 383–394 (2009)
82. Taly, A., Tiwari, A.: Switching logic synthesis for reachability. In: EMSOFT 2010, pp. 19–28. ACM, New York (2010)
83. Tarski, A.: A Decision Method for Elementary Algebra and Geometry. University of California Press, Berkeley (1951)
84. Tenenbaum, M., Pollard, H.: Ordinary Differential Equations. Dover Publications (October 1985)

85. Tomlin, C., Lygeros, J., Sastry, S.: A game theoretic approach to controller design for hybrid systems. Proceedings of the IEEE 88(7), 949–970 (2000)
86. Wang, S., Zhan, N., Guelev, D.: An assume/Guarantee based compositional calculus for hybrid CSP. In: Agrawal, M., Cooper, S.B., Li, A. (eds.) TAMC 2012. LNCS, vol. 7287, pp. 72–83. Springer, Heidelberg (2012)
87. Wildmoser, M., Nipkow, T.: Certifying machine code safety: Shallow versus deep embedding. In: Slind, K., Bunker, A., Gopalakrishnan, G.C. (eds.) TPHOLs 2004. LNCS, vol. 3223, pp. 305–320. Springer, Heidelberg (2004)
88. Wolfram: Mathematica Documentation, `http://reference.wolfram.com/mathematica/guide/Mathematica.html`
89. Xia, B.: DISCOVERER: a tool for solving semi-algebraic systems. ACM Commun. Comput. Algebra 41(3), 102–103 (2007)
90. Yang, L.: Recent advances on determining the number of real roots of parametric polynomials. J. Symb. Comput. 28(1-2), 225–242 (1999)
91. Yang, L., Xia, B.: Real solution classification for parametric semi-algebraic systems. In: Dolzmann, A., Seidl, A., Sturm, T. (eds.) Algorithmic Algebra and Logic, pp. 281–289 (2005)
92. Yang, L., Zhou, C., Zhan, N., Xia, B.: Recent advances in program verification through computer algebra. Frontiers of Computer Science in China 4, 1–16 (2010)
93. Zhan, N., Wang, S., Guelev, D.: Extending Hoare logic to hybrid systems. Tech. Rep. ISCAS-SKLCS-13-02, State Key Lab. of Computer Science, Institute of Software, Chinese Academy of Sciences (2013)
94. Zhao, H., Zhan, N., Kapur, D., Larsen, K.G.: A "hybrid" approach for synthesizing optimal controllers of hybrid systems: A case study of the oil pump industrial example. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 471–485. Springer, Heidelberg (2012)
95. Zhao, H., Zhan, N., Kapur, D., Larsen, K.G.: A "hybrid" approach for synthesizing optimal controllers of hybrid systems: A case study of the oil pump industrial example. CoRR abs/1203.6025 (2012), `http://arxiv.org/abs/1203.6025`
96. Zhou, C., Hansen, M.: Duration Calculus — A Formal Approach to Real-Time Systems. Monographs in Theoretical Computer Science. An EATCS Series. Springer, Heidelberg (2004)
97. Zhou, C., Hoare, C., Ravn, A.P.: A calculus of durations. Information Processing Letters 40(5), 269–276 (1991)
98. Zhou, C., Wang, J., Ravn, A.P.: A formal description of hybrid systems. In: Alur, R., Henzinger, T.A., Sontag, E.D. (eds.) HS 1995. LNCS, vol. 1066, pp. 511–530. Springer, Heidelberg (1996)
99. Zou, L., Lv, J., Wang, S., Zhan, N., Tang, T., Yuan, L., Liu, Y.: Verifying Chinese train control system under a combined scenario by theorem proving. In: Shankar, N. (ed.) VSTTE 2013. LNCS. Springer, Heidelberg (to appear, 2013)

Author Index