

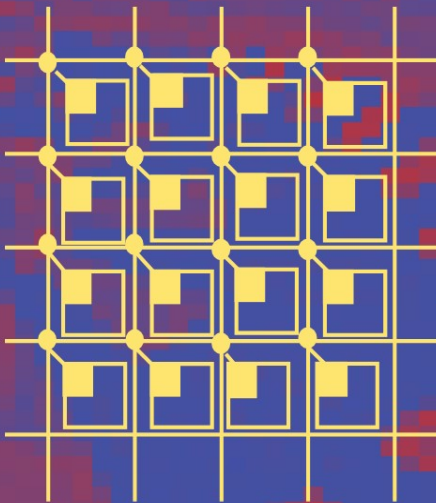
State-of-the-Art
Survey

LNCS 6957

Bernhard K. Aichernig
Frank S. de Boer
Marcello M. Bonsangue (Eds.)

Formal Methods for Components and Objects

9th International Symposium, FMCO 2010
Graz, Austria, November/December 2010
Revised Papers



 Springer

Commenced Publication in 1973

Founding and Former Series Editors:

Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

David Hutchison

Lancaster University, UK

Takeo Kanade

Carnegie Mellon University, Pittsburgh, PA, USA

Josef Kittler

University of Surrey, Guildford, UK

Jon M. Kleinberg

Cornell University, Ithaca, NY, USA

Alfred Kobsa

University of California, Irvine, CA, USA

Friedemann Mattern

ETH Zurich, Switzerland

John C. Mitchell

Stanford University, CA, USA

Moni Naor

Weizmann Institute of Science, Rehovot, Israel

Oscar Nierstrasz

University of Bern, Switzerland

C. Pandu Rangan

Indian Institute of Technology, Madras, India

Bernhard Steffen

TU Dortmund University, Germany

Madhu Sudan

Microsoft Research, Cambridge, MA, USA

Demetri Terzopoulos

University of California, Los Angeles, CA, USA

Doug Tygar

University of California, Berkeley, CA, USA

Gerhard Weikum

Max Planck Institute for Informatics, Saarbruecken, Germany

Bernhard K. Aichernig Frank S. de Boer
Marcello M. Bonsangue (Eds.)

Formal Methods for Components and Objects

9th International Symposium, FMCO 2010
Graz, Austria, November 29 - December 1, 2010
Revised Papers



Springer

Volume Editors

Bernhard K. Aichernig
Graz University of Technology
Institute for Software Technology
Inffeldgasse 16b
8010 Graz, Austria
E-mail: aichernig@ist.tugraz.at

Frank S. de Boer
Centre for Mathematics and Computer Science, CWI
Science Park 123
1098 XG Amsterdam, The Netherlands
E-mail: f.s.de.boer@cwi.nl

Marcello M. Bonsangue
Leiden University
Leiden Institute of Advanced Computer Science
P.O. Box 9512
2300 RA Leiden, The Netherlands
E-mail: marcello@liacs.nl

ISSN 0302-9743
ISBN 978-3-642-25270-9
DOI 10.1007/978-3-642-25271-6
Springer Heidelberg Dordrecht London New York

e-ISSN 1611-3349
e-ISBN 978-3-642-25271-6

Library of Congress Control Number:

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

LNCS Sublibrary: SL 2 – Programming and Software Engineering

© Springer-Verlag Berlin Heidelberg 2011

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer. Violations are liable to prosecution under the German Copyright Law.

The use of general descriptive names, registered names, trademarks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

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

Large and complex software systems provide the necessary infrastructure in all industries today. In order to construct such large systems in a systematic manner, the focus in development methodologies has switched in the last two decades from functional issues to structural issues: both data and functions are encapsulated into software units which are integrated into large systems by means of various techniques supporting reusability and modifiability. This encapsulation principle is essential to both the object-oriented and the more recent component-based software engineering paradigms.

Formal methods have been applied successfully to the verification of medium-sized programs in protocol and hardware design. However, their application to the development of large systems requires more emphasis on specification, modeling and validation techniques supporting the concepts of reusability and modifiability, and their implementation in new extensions of existing programming languages like Java.

The 9th Symposium on Formal Methods for Components and Objects (FMCO 2010) was held in Graz, Austria, from November 29 to December 1, 2010. The venue was Hotel Weitzer. FMCO 2010 was realized as a concertation meeting of European projects focussing on formal methods for components and objects. This volume contains 20 revised papers submitted after the symposium by the speakers of each of the following European projects involved in the organization of the program:

- The FP7-IST project AVANTSSAR on automated validation of trust and security of service-oriented architectures. The contact person is Luca Viganò (University of Verona, Italy).
- The FP7-IST project DEPLOY on industrial deployment of advanced system engineering methods for high productivity and dependability. The contact person is Alexander Romanovsky (Newcastle University, UK).
- The ESF-COST Action IC0701 on formal verification of object-oriented software. The contact person is Bernhard Beckert (Karlsruhe Institute of Technology, Germany).
- The FP7-IST project HATS on highly adaptable and trustworthy software using formal models. The contact person is Reiner Hähnle (Chalmers University of Technology, Sweden).
- The FP7-SST project INESS on an integrated European railway signaling system. The contact person for work relating to FMCO is Jim Woodcock (University of York, UK).
- The FP7-IST project MADES on a model-driven approach to improve the current practice in the development of embedded systems. The contact person is Alessandra Bagnato (TXT e-solutions, Italy).

- The FP7-IST project MOGENTES on model-based generation of tests for dependable embedded systems. The contact person for work relating to FMCO is Bernhard Aichernig (Graz University of Technology, Austria).
- The FP7-IST project MULTIFORM on integrated multi-formalism tool support for the design of networked embedded control systems. The contact person for work relating to FMCO is Christian Sonntag (TU Dortmund, Germany).
- The FP7-IST project QUASIMODO on quantitative system properties in model-driven design of embedded systems. The contact person is Kim G. Larsen (Aalborg University, Denmark).

The proceedings of the previous editions of FMCO have been published as volumes 2852, 3188, 3657, 4111, 4709, 5382, 5751, and 6286 of Springer's *Lecture Notes in Computer Science*. We believe that these proceedings provide a unique combination of ideas on software engineering and formal methods which reflect the expanding body of knowledge on modern software systems.

Finally, we thank all authors for the high quality of their contributions, and the reviewers for their help in improving the papers for this volume.

July 2011

Bernhard K. Aichernig
Frank de Boer
Marcello Bonsangue

Organization

FMCO 2010 was organized by the Institute for Software Technology, Graz University of Technology, Austria, in collaboration with the Centrum voor Wiskunde en Informatica (CWI), Amsterdam, and the Leiden Institute of Advanced Computer Science, Leiden University, The Netherlands.

Program Organizer

Bernhard K. Aichernig Graz University of Technology, Austria

Steering Committee

Frank de Boer	CWI and Leiden University, The Netherlands
Marcello Bonsangue	Leiden University, The Netherlands
Stefan Hallerstede	University of Düsseldorf, Germany
Michael Leuschel	University of Düsseldorf, Germany
Eric Madelaine	INRIA Méditerranée, France

Local Organization at Graz University of Technology

Bernhard K. Aichernig (Chair)
Harald Brandl
Arabella Gaß
Elisabeth Jöbstl
Petra Pichler
Herbert Pöckl
Stefan Tiran
Benedict Wright

Sponsoring Institutions

European FP7 project MOGENTES
Graz University of Technology
Graz Convention Bureau

Table of Contents

The AVANTSSAR Project

ASLan++ — A Formal Security Specification Language for Distributed Systems	1
<i>David von Oheimb and Sebastian Mödersheim</i>	
Orchestration under Security Constraints	23
<i>Yannick Chevalier, Mohamed Anis Mekki, and Michaël Rusinowitch</i>	
Customizing Protocol Specifications for Detecting Resource Exhaustion and Guessing Attacks	45
<i>Bogdan Groza and Marius Minea</i>	

The ESF Cost Action IC0701

Improving the Usability of Specification Languages and Methods for Annotation-Based Verification	61
<i>Bernhard Beckert, Thorsten Bormer, and Vladimir Klebanov</i>	
Program Specialization via a Software Verification Tool	80
<i>Richard Bubel, Reiner Hähnle, and Ran Ji</i>	

The DEPLOY Project

Model-Based Analysis Tools for Component Synthesis	102
<i>Luigia Petre, Kaisa Sere, and Leonidas Tsiopoulos</i>	
Shared Event Composition/Decomposition in Event-B	122
<i>Renato Silva and Michael Butler</i>	

The HATS Project

ABS: A Core Language for Abstract Behavioral Specification	142
<i>Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen</i>	
A Component Model for the ABS Language	165
<i>Michaël Lienhardt, Ivan Lanese, Mario Bravetti, Davide Sangiorgi, Gianluigi Zavattaro, Yannick Welsch, Jan Schäfer, and Arnd Poetzsch-Heffter</i>	

Compositional Algorithmic Verification of Software Product Lines 184
Ina Schaefer, Dilian Gurov, and Siavash Soleimanifard

Variability Modelling in the ABS Language 204
Dave Clarke, Radu Muschevici, José Proença, Ina Schaefer, and Rudolf Schlatte

The INESS Project

Automated Verification of Executable UML Models 225
Helle Hvid Hansen, Jeroen Ketema, Bas Luttik, MohammadReza Mousavi, Jaco van de Pol, and Osmar Marchi dos Santos

Verification of UML Models by Translation to UML-B 251
Colin Snook, Vitaly Savicks, and Michael Butler

The MADES Project

Towards the UML-Based Formal Verification of Timed Systems 267
Luciano Baresi, Angelo Morzenti, Alfredo Motta, and Matteo Rossi

The MOGENTES Project

Generic Fault Modelling for Fault Injection 287
Rickard Svenningsson, Henrik Eriksson, Jonny Vinter, and Martin Törngren

Tightening Test Coverage Metrics: A Case Study in Equivalence Checking Using *k*-Induction 297
Alastair F. Donaldson, Nannan He, Daniel Kroening, and Philipp Rümmer

The MULTIFORM Project

The Hierarchical Compositional Interchange Format 316
Damian Nadeles Agut, Bert van Beek, Harsh Beohar, Pieter Cuijpers, and Jasper Fonteijn

Application of Model-Checking Technology to Controller Synthesis 336
Alexandre David, Jacob Deleuran Grunnet, Jan Jakob Jessen, Kim Guldstrand Larsen, and Jacob Illum Rasmussen

The QUASIMODO Project

Testing Real-Time Systems under Uncertainty	352
<i>Alexandre David, Kim Guldstrand Larsen, Shuhao Li, Marius Mikucionis, and Brian Nielsen</i>	
Model-Checking and Simulation for Stochastic Timed Systems	372
<i>Arnd Hartmanns</i>	
Author Index	393

ASLan++ — A Formal Security Specification Language for Distributed Systems

David von Oheimb¹ and Sebastian Mödersheim²

¹ Siemens Corporate Technology, IT Security, Munich, Germany

David.von.Oheimb@siemens.com, ddvo.net

² DTU Informatics, Technical University of Denmark, Lyngby, Denmark

samo@imm.dtu.dk, imm.dtu.dk/~samo

Abstract. This paper introduces ASLan++, the AVANTSSAR Specification Language. ASLan++ has been designed for formally specifying dynamically composed security-sensitive web services and service-oriented architectures, their associated security policies, as well as their security properties, at both communication and application level.

We introduce the main concepts of ASLan++ at a small but very instructive running example, abstracted from a company intranet scenario, that features non-linear and inter-dependent workflows, communication security at different abstraction levels including an explicit credentials-based authentication mechanism, dynamic access control policies, and the related security goals. This demonstrates the flexibility and expressiveness of the language, and that the resulting models are logically adequate, while on the other hand they are clear to read and feasible to construct for system designers who are not experts in formal methods.

Keywords: services, security, specification language, formal analysis.

1 Introduction

Formal Security Analysis. Security in distributed systems such as web services and SOA is very difficult to achieve, because often the security problems are very subtle. Even systems that are simple to describe (such as the famous three-line Needham-Schroeder Public Key protocol) may have weaknesses that go unnoticed for years even when the system has been carefully studied [9]. Formal specification and verification of such systems can help to uncover weaknesses before they can be actually exploited. Especially automated verification tools can help to find the needle in the haystack — one trace of the system that violates the security goals among an enormous number of traces that are fine.

Over the last decade, formal verification for security has made a lot of progress. In the late 90s, automated protocol verification tools began to emerge that focussed on simple security protocols that can be described by an exchange of messages (e.g., in Alice&Bob-style notation). Despite being small systems, their verification is very challenging, in particular considering that an intruder has an

unbounded choice in constructing messages, which may involve algebraic properties of the cryptographic primitives. Moreover one cannot give a bound on the number of sessions that can be executed in parallel. These problems are now well understood, both theoretically in terms of complexity and decidability [21,12,15], and in terms of methods and tools that are practically feasible automated verification [8,1,13,16].

Limitations of Security Protocol Analysis. The focus of simple security protocols is however quite limited, ignoring a lot of aspects that play a crucial role in distributed systems and that often *are* relevant for security.

The first very common aspect that falls out of the simple structure of security protocols is non-linear communication. For instance, a (web-) server typically listens for requests that must be in one of several types of formats; depending on the request, the server will start an appropriate workflow, possibly contacting other servers that implement subtasks of the workflow, and then finally give a response to client who sent the initial request.

This brings us immediately to a second aspect: the described transaction may sometimes *not* be independent from all other transactions, but for instance may be related via dynamic distributed state. For instance, in case of an online shop, a database maintained by the server may contain the set of all processed orders and their status, the set of all registered customers, and other related information. Processing different requests may depend on this database, for instance a registered user can send a request to see all her recent orders — provided the user can authenticate herself by username and password or maybe by a cookie. Another subsequent request could then be to cancel or change an order that has not yet been shipped. These aspects are completely outside the realm of simple security protocols where different sessions are essentially independent and the only information shared between different sessions are static long-term keys.

A third important aspect concerns the relation to dynamic security policies. For example, when a server receives a request from a client to access a resource it controls, it may need to check whether the particular client has the necessary access rights. These access rights may not be static but may for instance depend on who is a member of the group that owns a particular resource, and these memberships may change over the time. The change of group memberships may itself be transactions of the system that is again governed by some access control policies, e.g., only members of a certain role, say *manager*, are authorized to change group memberships.

AVANTSSAR and Its Specification Language. The EU-funded Project AVANTSSAR has been concerned with developing a formal specification language and automated verification methods and tools to handle systems at design level in which all these three aspects are relevant: non-linear work-flow, relationships between workflows (for instance via databases), and access control policies. In this paper, we describe the AVANTSSAR Specification Language

ASLan++ [4], which has been developed as a joint effort by the partners of the project. The design goal of ASLan++ were

1. expressiveness sufficient to describe the security-relevant aspects of service-oriented architectures as described above,
2. ease of use for systems designers, in particular being close to the way designers think about and describe such systems, allowing to abstract from details whenever they are not relevant or can be “factored out”,
3. compatibility with existing and emerging verification methods so that automatically analyzing specifications is feasible at least for small number of parallel processes, without being biased to a particular method.

Structure of This Paper. In this paper, we discuss the main concepts of ASLan++ and how they can be used for modeling the most relevant aspects of service-oriented architectures. We also briefly discuss the rationale behind some design decisions and some consequences for the verification methods, drawing in particular from our experience in modeling a number of larger case studies.

For introducing ASLan++, we employ a small but very instructive running example specification. Its full text may be found in the appendix, while in the following sections we describe it piece-by-piece, progressing from basic structuring of the specification over its procedural aspects to security policies, communication properties, and security goals.

The example describes part of a company intranet scenario. Employees may access files according to a dynamic access control policy. A central server keeps track of the access rights. Both managers and employees can influence the policy.

2 Specification Structure and Execution

2.1 Specifications

An ASLan++ specification of a system and its security goals consists of a hierarchy of *entities*. An entity may import other entities contained in separate files, which in turn contain a hierarchy of entity declarations. The top-level entity, usually called **Environment**, serves as the global root of the system being specified, similarly to the “main” procedure of a program. In our example specification, the **Environment** has two sub-entities: **Session** and **Employee**, where the former has in turn two sub-entities: **Server** and **Manager**. The **Manager** entity, for example, is used to describe the behavior of any honest manager as well as the security requirements that can be stated from her perspective.

2.2 Entities and Agents

Entities are the major ASLan++ building blocks, which are similar to classes in Java or roles in HPSL [10] and other security protocol specification languages. Entities are a collection of declarations and behavior descriptions. They can have parameters and local variables, with the usual nested scoping w.r.t. sub-entities.

Entities are like blueprints that can be instantiated to any number of processes (or threads), each executing the *body* of the entity. With the exception of sets, the parameters of an entity have call-by-value semantics: the entity obtains a copy of the values and may change them without side-effects for the “calling” process. On the other hand, one can easily model shared data [4, §2.4].

Each entity has an explicit or implicit formal parameter **Actor**, which is similar to **this** or **self** in object-oriented programming languages. The value of **Actor** is the name of the agent playing the role defined by the entity. This is important for defining the security properties of the entity.

The entity and instance structure of our example is as follows, where entity and variable names are uppercase, while constant and type names are lower-case. ASLan++ comments start with a “%” symbol and extend until end of the line.

```
entity Environment {
  ...
  entity Session (M, S: agent) {
    entity Server(M, Actor: agent) {
      ...
    }
    entity Manager(Actor, S: agent) {
      ...
    }
    body { % of Session
      ...
      new Server (M,S);
      new Manager(M,S);
    }
  }
  entity Employee(Actor, S: agent) {
    ...
  }
  body { % of Environment
    ...
    any M. Session(M,centralServer);
    new Employee(e1,centralServer);
    new Employee(e2,centralServer);
  }
}
```

There is (at least) one instance of entity **Session**, each invoked by a statement `any M. Session(M,centralServer)` (described later). It has two formal parameters (of type **agent**): **M** refers to the agent playing the manager role, while **S** holds the name of the server. Each session launches in parallel a thread of the **Server** and a **Manager** instance, by statements like `new Manager (M,S)`. The session(s) runs in parallel with two (or more) **Employee** instances.

The **Manager** entity has two parameters: **Actor** is used to refer to herself, while **S** holds the name of the server she is going to interact with. The parameters of the **Employee** entity are analogous to **Manager**.

Instances of the **Server** entity will actually obtain the name of the manager via the manager’s messages described below. Still, for the sake of relating entities for the security goals, we need to give **M**, the variable that will hold the manager’s agent name, as a formal parameter of **Server**. The other parameter of **Server** is, as usual, the **Actor**.


Note that while each instance of `Manager` and `Employee` (typically) has a different agent playing the respective role, for example referred to by the constants `e1` and `e2` used for employees, there is just a single constant `centralServer` used as actor of the `Server` entity. This is how we model that the server is global.

2.3 Execution Model

The instantiation of an entity is *in parallel*: the caller starts a new process that runs in parallel to the caller. A subtle point is the granularity at which parallel processes can be interleaved. Consider that a web server may make quite a number of intermediate computations between receiving a request and sending a reply. Running in parallel with other processes (e.g., other instances of the same server that currently serve a different request) produces an exponential number of interleavings, which is difficult to handle for many verification methods. There is also a number of classical problems attached, e.g., if we think of a two threads of the server checking and modifying the database, this can easily lead to race conditions. For ASLan++ we have chosen a particular way to deal with interleavings. Whenever an entity receives a message and then acts upon that, we consider its subsequent activity *atomic* up to the point where the entity goes back into a state of waiting for further messages. The reason is quite pragmatic: we get a coarse interleaving model that is feasible for verification tools without the user having to code tool-related optimizations into the specification (i.e., declaring atomicity-blocks to help the tools). At the same time, this can be regarded as a reasonable model for many situations: when the server’s computation is related to a shared resource, e.g., reading from a database and then writing a change into the database, it is clear that in the implementation that process should get a lock on the server so that other processes do not change the database in between. ASLan++ thus allows to abstract from such locking mechanisms, and in fact they are often not the focus of a security verification. However, if desired, ASLan++ also allows to declare “custom breakpoints” (overriding the default atomicity behavior) to model a finer interleaving model.

ASLan++ offers experimental support for constraints on the global system run via LTL formulas, which may be used to specify e.g., fairness assumptions.

2.4 Dishonest Agents and the Intruder

The attacker is known as the *intruder* and can be referred to by the constant `i` (of type `agent`). Yet we allow the intruder to have more than one “real name” . To this end, we use the predicate `dishonest` that holds true of `i` and of every pseudonym (i.e., alias name) `A` of `i`.

¹ The intruder may have several names that he controls. This reflects a large number of situations, like an honest agent who has been compromised and whose long-term keys have been learned by the intruder, or when there are several dishonest agents who collaborate. This worst case of a collaboration of all dishonest agents may be simply modeled by one intruder who acts under different identities.

As long as the actual value of the `Actor` parameter of an entity is an honest agent, the agent faithfully plays the role defined by the entity. If the `Actor` parameter value is dishonest already on instantiation of the entity, which is typically the case for some of the possibilities included in symbolic sessions (cf. [subsection 2.6](#)), the body of the entity is ignored because the intruder behavior subsumes all honest and dishonest behavior.

We also allow that an entity instance gets compromised later, that is, the hitherto honest agent denoted by the `Actor` of the entity becomes dishonest. Once an agent has become dishonest, for instance because it has been corrupted, it can never become honest again.

2.5 Declarations

An entity may contain declarations of types, variables, constants, functions, macros, (Horn) clauses, and algebraic equations.

Unless declared non-public, constants and functions are public, such that the intruder knows them and thus may (ab-)use them. Moreover, function symbols are by default interpreted in the free term algebra (modulo algebraic equations), such that they are by default invertible in each argument. This conveniently reflects the typical behavior of message constructors, like the ones declared in our example:

```
login (agent , symmetric_key): message;
changeGroup (agent , agent set , agent set): message;
assignDeputy(agent): message;
requestAccess(file): message;
grantedAccess(file): message;
deniedAccess(file): message;
```

where the types in parentheses specify their argument types.

Message constructors abstract from the actual implementation details of how messages are actually encoded. Essentially the only property we rely on is their invertibility, such that e.g., the intruder may obtain `A`, `G1`, and `G2` from knowing `changeGroup(A, G1, G2)`. Since often a function application term is better readable when the first argument is written before the function symbol, ASLan++ offers syntactic sugar for this, such that we can equivalently write in “object-oriented style”: `A->changeGroup(G1, G2)`. The message constructors just mentioned, as well as the remaining symbols declared in the global `symbols` section, will be described in more detail below where appropriate.

Types may have subtypes, e.g., the (built-in) relation `agent < message` means that any value of type `agent` may be used in a context where a value of type `message` is expected. The type `message` includes all those values that may be sent over the network, in particular concatenation `M1.M2` and tuples `(M1, M2)` of sub-messages `M1` and `M2`. For “atomic” values in messages, one may use the subtype `text`, which may be dealt with more efficiently during model-checking. For instance, we declare an abstract type of files (or better: file identifiers) as

```
types
  file < text;
```


Sets, which are passed by reference, are not a subtype of `message`, such that they cannot be directly sent as messages.² Sets and tuples have parameters for their element types, e.g., `nat set` and `agent * message`).

Symbols may also be declared in the respective sections of the various entities, in particular the local variables that their instances use internally. For instance, both `Manager` and `Server` declare

```
symbols
  Cookie: cookie;
```

where `cookie` is a subtype of `text`.

2.6 Statements

Statements may be the usual assignments, branches and loops, but also non-deterministic selections, assertions, generation of fresh values and of new entity instances, transmission of messages (i.e., send and receive operations), and introduction or retraction of facts, which represent state-dependent truth values.

The `select` statement is typically used within the main loop of a server, as it is the case in our example for the `Server` entity:

```
body {
  while(true) {
    select {
      on(... & ... ): {
        ...
      }
      ...
      on(?A *->* Actor: requestAccess(?F)): {
        ...
      }
    }
  }
}
```

Such a statement handles a variety of potential incoming requests or other events such as timeouts. It checks the guards given, blocking as long as no guard is fulfilled, then nondeterministically chooses any one of the fulfilled guards and executes the corresponding statement block. The evaluation of the chosen guard assign variables that are written with the `?` symbol before their name. For instance, the guard

```
on(?A *->* Actor: requestAccess(?F)): { ... }
```

(where in this context the decorated arrow symbol `*->*` denotes a communication channel with certain properties, as we will describe in [section 4](#)) can fire when a `requestAccess` message has been received from any authenticated agent `A` for any file `F`. When this guard is chosen, the values of these two variables are set according to the actual values received. Then in response the compound statement enclosed by the brackets `{ ... }` is executed.

Entity generation, introduced by the keyword `new` or `any`, instantiates sub-entities. This is only allowed for direct sub-entities, such that static and dynamic

² In [\[4\] §2.5](#)], we describe several possibilities to communicate sets.

scoping coincide. In our example, the `Session` entity creates new instances of the server and the `Manager` entity:

```
new Server (M,S);
new Manager (M,S);
```

These run in parallel and in this case happen to obtain on creation the same actual parameter values, `M` and `S`.

Symbolic entity generations, introduced by `any`, are a convenient shorthand for loosely instantiating an entity, in the following sense: the bound parameters of the entity, as indicated by the given list of variables, allows to explore all possible values, from the domain of their type (which may be any subtype of `message`). An optional guard, which may refer to the variables listed, constrains the selection. This mechanism is typically used to produce so-called *symbolic sessions*, where the bound variables range over type `agent`, such that (unless further constraints exist) their values include `i`, the name of the intruder.

In our example, we symbolically instantiate the `Session` entity by

```
any M. Session(M, centralServer);
```

Note that since we did not constrain the agent value for `M`, it may be in fact the intruder. The model checkers will use this freedom to look for attacks for both honest and dishonest instantiations for `M`.

2.7 Terms

Terms may contain variables (e.g., `A`), constants (e.g., `e1`), and function applications (to be more precise: function symbols applied to first-order terms, e.g., `requestAccess(F)`) including infix right-associative message concatenation, e.g. `M1.M2`) and tupeling (e.g., `(A,b2,0)`). Set literals are written as usual (e.g., `{A,B,C}`), while the basic operator on sets is the `contains` function, where the presence of the fact `Set->contains(X)` means that `X` is a member of `Set`.

3 Policies and Transitions

ASLan++ provides an extremely powerful way to specify security policies and their interaction with the dynamic system defined by the entities given in the specification. For simplicity, let us refer to the latter system in the following simply as *the transition system*. Policies are specified by a set of Horn clauses, e.g., stating that a person can get access to some resource if certain conditions are met. In our running example, there are only two such rules:

```
clauses
accessDirect(A,G,F): A->canAccess(F) :- G->isOwner(F) & G->contains(A);
accessDeputy(A,B,F): A->canAccess(F) :- A->deputyOf(B) & B->canAccess(F);
```

These rules make use of the following user-declared predicate symbols:

```
canAccess(agent      ,file): fact;
isOwner  (agent set,file): fact;
deputyOf (agent      ,agent): fact;
```

3.1 Predicates and Facts

Instead of the usual type *bool* for truth values, ASLan++ uses the type **fact**. Terms denoting atomic propositions, generally known as *predicates*, are represented by functions with result type **fact**. The question whether an atomic proposition holds or not is expressed by the (non-)existence of the respective predicate term in a global “fact space”. Facts may be combined with the usual logical operators in LTL formulas to express goals, and they are also used in conditions (in **if**, **while**, and **select** statements) known as *guards*.

By default a fact does not hold, but it may be explicitly introduced (simply by writing it as an ASLan++ statement) and retracted. The constant **true** is introduced automatically, while the constant **false** is never introduced. Facts may also be generated by Horn clauses, as described next.

3.2 Horn Clauses

The first above rule says that an agent A can access a file F if A is a member of a group G that is the owner of F . The second rule says that A can access file F if A is a deputy of another agent B who has access to F . Note that it is only for the sake of simplicity of the example that this latter rule models a “complete delegation” of all access rights while most real systems would make more fine-grained delegations.

The symbols A , B , G , F are variables that can be instantiated with arbitrary values and hence are regarded as “parameters” of the rules; this allows in the output of (attack) traces to clearly announce which rule with which values of the parameters had been applied. Note that the second rule is “recursive”: if A is the deputy of B and B is the deputy of C , then A also gets access to everything that C has access to — and such a line of deputies can be extended ad libitum, implying delegation of access rights along this line.

It is important to see that these rules are positive formulations of access control conditions: A gets access to a file F if and only if there is *some* way to derive $A \rightarrow \text{canAccess}(F)$ with the Horn clauses. We do not allow negative formulations such as “ A does *not* get access if ...”. This has the advantage that ASLan++ policies can never be *inconsistent* in the sense that one rule allows access while another one would deny it. The price that we pay for this is that it is harder in ASLan++ to formulate a higher-level policy that overrides the judgements of a lower level policies; we discuss this below. Observe that by allowing arbitrary definite first-order logic Horn clauses, this alone gives a Turing-complete programming language (namely a subset of Prolog)³. This expressivity implies that derivability in ASLan++ policies is in general undecidable. There are several ways to restrict this concept to decidable fragments, e.g., allowing only primitive recursion. It was part of the language design of ASLan++ *not* to commit to such a particular restricted fragment, which may be specific to a verification

³ There is even some (experimental, so far) support for (in-)equalities as side conditions on the right-hand side of clauses.

method. Our method thereby allows to formulate policies in very different ways, e.g., SecPAL and DKAL policies [6,18] can be specified.

It is crucial to first distinguish two kinds of facts, namely the *state facts*: those explicitly introduced by the transition system, and *policy facts*: those more implicitly “generated” by Horn clauses. In our example, `canAccess` is the only policy fact, because it is the only fact that can be produced by the policy rules. All the other facts are state facts. We will come back why we must insist on this distinction.

3.3 Policy Interaction

There are now two ways how the policies can interact with the transition system that we describe by the ASLan++ entity specifications and their instantiations.

First, transitions of an entity can depend on the judgement of policies. For our example, consider the transaction where an authenticated user requests access to a file: the server governing file access should first check whether the policy actually allows this user access to the requested file. Here is the code snippet of the server’s behavior (thus `Actor` is `centralServer` here):

```
on(?A *->* Actor: requestAccess(?F)): {
  if (A->canAccess(F))
    Actor *->* A: grantedAccess(F);
  else
    Actor *->* A: deniedAccess(F);
}
```

The response of the server, either `grantedAccess(F)` or `deniedAccess(F)`, depends on whether `A->canAccess(F)` holds, which is determined by the two Horn clauses as explained above.

The second way that policies can interact with the transition system is just the other way around: the transition system can generate and retract state facts on which the Horn clauses depend. For instance, there can be transitions that change who is the owner of a file, or who is member of which group or who is deputy of whom, and this has an immediate effect on the access rights via the rules. In our example, let us consider that a manager M can tell the server that a certain employee A changes from a group G_1 to a group G_2 , so that the server updates the group membership information. Here is the code snippet from the point of view of the server (i.e., `Actor`):

```
on(M *->* Actor: (?A->changeGroup(?G1,?G2)) & ?G1->contains(?A)): {
  retract(G1->contains(A));
  G2->contains(A);
}
```

Like with the messages sent by an employee, here the manager’s command is transmitted on a secure channel (including authentication of the manager), and again the command is abstracted into the message constructor `changeGroup` that has the relevant information (the agent A that changes group, and the source and destination group) as parameters. The server just *retracts* the fact that A is a member of G_1 and *introduces* the fact that G_2 now contains A . Note the command is simply ignored if A is not a member of group G_1 at the time the

command is received; in a more detailed model, one would include a feedback message (whether the command was accepted or not) to the manager.

3.4 Concrete Policy Example

Let us consider the consequences of the transition just described for our policy. For concreteness, let us consider a state where we have a manager m_1 , three employees e_1, e_2 and e_3 , and two groups $g_1 = \{e_1, e_2\}$ and $g_2 = \{e_3\}$. Consider moreover files f_1, f_2 , where group g_i owns file f_i , and that initially there are no deputy relations. All this is formulated by the following contents of the Environment declaration:

```

symbols % for the concrete access examples
m1: agent;
e1, e2, e3: agent;
g1, g2: agent set;
f1, f2: file;
...

body { % of Environment
% for the concrete access examples:
m1->isManager;
g1->contains(e1); g1->contains(e2);
g2->contains(e3);
g1->isOwner(f1);
g2->isOwner(f2);
...
}

```

By our access control rules, e_1 and e_2 can access f_1 and e_3 can access f_2 .

When a manager successfully issues the command `e1->changeGroup(g1,g2)`, this implies that e_1 loses her or his access to f_1 but gains access to f_2 . Thus, the access rights are obtained as the *least closure* of the state facts under the policy rules: everything that can be derived from the current state by the policy is true, everything else is false.

To illustrate the effects of state transitions to the policy in more depth, let us consider another transaction where A assigns B as her deputy:

```

on(?A *->* Actor: assignDeputy(?B)): {
  B->deputyOf(A);
}

```

Consider that in this way e_1 becomes deputy of e_2 while both are still in group g_1 . If the transfer of e_1 from group g_1 to g_2 is performed in this situation, e_1 gets access to f_2 , but it does not lose the access to f_1 . This is because access to f_1 is still derivable through the deputy relation: e_1 has access to everything that e_2 has access to (via the second policy rule), and e_2 is still a member of g_1 and thus has direct access to f_1 (via the first policy rule).

This illustrates how expressive the combination of transitions and policies actually is. In particular, there can be several independent reasons why an agent has access to a particular resource. Each of these reasons can change dynamically when people enter or leave groups, become deputies of others or stop being deputies. If one reason for access is removed by a transition, but another reason remains, then also the access right remains. Once all reasons are removed, also

the access right is gone. In the previous example, if e_1 stops being deputy of e_2 (say, because e_2 returns from vacation, which can be modeled by a straightforward `revokeDeputy` command) then with that also the reason for access to f_1 is removed, and since no other reason is left, e_1 no longer has access to f_1 .

3.5 Meta Policies

Of course this example has been deliberately kept simple, but let us now review briefly how certain more complex aspects can be modeled. One may model the hierarchical structure in a company and model that one inherits the access rights of one's subordinates:

```
accessSuperior(A,B,F): A->canAccess(F) :- A->superiorOf(B) & B->canAccess(F);
superiorDirect(A,B) : A->superiorOf(B):- A->managerOf(B);
superiorTrans (A,B,C): A->superiorOf(C):- A->superiorOf(B) & B->superiorOf(C);
```

This shows a different application of the Policy/Horn clauses: mathematically speaking, we define the relation `superiorOf` as the *transitive closure* of the `managerOf` relation. Intuitively, `managerOf` gives the direct superior and is a relation controlled by the transition system just like the other state facts like `deputyOf` etc.; while `superiorOf` yields all superiors over any number of hierarchy levels, and this is “immediately computed” depending on the state of `managerOf`.

This example of “superiors can access everything that their subordinates can access” can be regarded as a *meta policy*, i.e., actually a policy about policies or giving boundaries to policies. This is increasingly important because policies may be expressed (formally) at different levels, e.g., there may be policies at the level of workgroups or divisions of a company, or at the level of the company itself, or on top of that policies required by governmental law.

We have just seen an example of a positive top-level policy, which is easy to integrate. More difficult are negative top-level policies. Take the following negative meta policy as an example: one cannot assign a deputy outside one's own group. This aims at preventing the situation in the above example where e_1 still has access to a file of his old group because he is deputy of an old group member e_2 . We cannot directly formulate such negative conditions in the Horn clauses of ASLan++, but we could code it indirectly into the transition for assigning deputies:

```
on(?A **>* Actor: assignDeputy(?B) & ?G->contains(?A) & ?G->contains(?B)): {
  B->deputyOf(A);
}
```

Here the first condition `G->contains(A)` determines *one* group $?G$ that A is member of — in fact we have not explicitly enforced that every agent is member of at most one group — and the second condition requires that the to-be-assigned deputy B is also member of the same group G . However, this only enforces that at the moment of deputy assignment, A and B are member of one common group, and in fact the high-level policy is violated as soon A or B change to another group while the deputy relation is in place. In fact a real system may be built like this and have the weakness that the meta policy is not checked

when people change groups. We thus see it as a strength of ASLan++ that such systems (with all their flaws) can be modeled and the problem be discovered by automated verification.

To formalize a system that realizes the deputy-in-same-group meta policy (no matter how), the easiest way is to actually allow in the model deputies outside the group, but to enforce the same-group constraints whenever access is granted on grounds of the deputy relation, i.e., refining our original `accessDeputy` rule:

```
accessDeputy(A,B,F,G) : A->canAccess(F) :- A->deputyOf(B) & B->canAccess(F)
& G->contains(A) & G->contains(B);
```

If there are other decision made based on the `deputyOf` relation, they would have to be refined similarly.

4 Channels

A very typical aspect of the systems we model in ASLan++ is that they are distributed and communicate over (initially) insecure channels that could be accessible to an intruder who may read, intercept, insert and modify messages. It is also common to secure the communication lines by protocols like TLS or IPSec and thereby obtain a virtual private network, i.e., as if the distributed components were directly connected by secure lines.

4.1 Abstraction Levels

ASLan++ is of course expressive enough to directly model protocols like TLS and IPSec, using the classical cryptographic primitives for encryption and digital signatures, but this is not really desirable: one should not model large systems monolithically and in all detail, but, whenever possible, distinguish different layers and components in a system. This approach entails to verify high-level applications that are run over secure channels independently of the low-level protocol that provides these channels. This gives also greater significance to the verification result: the application is then secure even when *exchanging* the low-level secure channel protocol. Vice-versa, the channel protocol should be verified independently of a concrete application, so that it can be used for other applications as well. There are first results for such *compositional reasoning* for channels [2011].

ASLan++ supports an abstract notion of channels where we simply state that messages are transmitted under certain assumed security properties. We have already seen examples above:

```
?A *->* Actor: requestAccess(?F)
```

The stars mean that the respective side of the channel is protected. Protection on the receiver side means confidentiality: it can only be received by the intended receiver. Protection on the sender side means authentication: it is certain that the message indeed comes from the claimed sender. Authentication also includes that the intended recipient is part of what is being authenticated; so the receiver

can see whether this message was really intended for him (even though everybody can read it when confidentiality is not stipulated). This follows the definition of the *cryptographic channel model (CCM)* and the *ideal channel model (ICM)* presented in [20]. There is an alternative notation supporting the *abstract channel model (ACM)* of [2]; we leave this out here for lack of space and also because the integration into ASLan++ is not finished at the state of this writing. Much more detail on the three channel models may be found e.g., in [4, §3.8].

The channels we have in our little example only ensure confidentiality and authenticity/integrity, they do not incorporate other properties such as recency (which can be achieved using the \rightarrow), disambiguation of different channels between the same principals (which can be achieved by including distinguishing channel/session identifiers in the messages) or the ordering of messages (which can be achieved by including sequence numbers). For details on these aspects of channel modeling we refer to [4, §2.9].

4.2 Client Authentication

We do want to illustrate however one very common situation in modeling channels: one side may not be authenticated. The most typical example is TLS where usually the server possesses a certificate, but the client does not. One may model this situation by declaring transmissions from client to server as being only confidential (but not authentic) and transmissions from server to client as only authentic (but not confidential). However, TLS with an unauthenticated client provides actually more security guarantees, namely sender and receiver invariance for the client: even though the client's request is not authenticated, the response from the server is sure to go only to that client who posed the request, and subsequent requests can be associated to the same client. [20] suggests regarding this as a secure channel except that the client is not authenticated by its real name but by a self-chosen pseudonym. We denote this such a channel in ASLan++ as `[Client]_ [Pseudonym] *->* Server`.

Such unilaterally authenticated channels are of high relevance in practice for many applications, such as transmitting authentication information of the client like passwords or cookies to the server. In this case, the server can definitely link the pseudonym to the client's real name. If we wish to just abstract from the TLS channel, but not from the authentication mechanism that is run over the channel, then the pseudonymous channel notation of ASLan++ gives us the possibility to do so. Let us consider for that reason a refinement of our previous example. Before, we used a secure channel between a Manager M and the server S . Let us now model that M has a TLS channel to S where M is not authenticated. As a first step, the manager would send a login with his name and password. The password we model as symmetric key which is a function of M and S :

```
nonpublic noninvertible password(agent, agent): symmetric_key;
```

Here, `nonpublic` means that no agent itself can apply the password function, one can only initially know passwords or learn them during a message transmission. Similarly `noninvertible` means that one cannot obtain the agent names from

a given password. We ignore here bad passwords, but we explicitly allow the intruder to have its own password with S , namely $\text{password}(i,S)$, which is initially known to the intruder.

We model that when a manager logs in to the server (here the **Actor**) over the pseudonymous channel $[?M]_{[?MP]}$ $*\rightarrow*$ **Actor**, the server creates a cookie for that manager, sends it back on the pseudonymous channel, and stores the cookie, along with the manager's identity, in a cookie database:

```
on([?M]_{[?MP]} *\rightarrow* Actor: login(?M,password(?M,Actor)) & ?M->isManager): {
  Cookie := fresh();
  cookies(Actor)->contains((M,Cookie));
  Actor *\rightarrow* [M]_{[MP]}: Cookie;
}
```

Note that in all transactions involving a manager we write $?M$ and $?MP$ because the identity of M and her pseudonym $?MP$ are learned by the server at this point. This allows for modeling multiple managers. When a manager connects, stating its own name M , the server requires an abstract login message that consists of the user name M and password. After these have been verified, it makes sense to check if M is indeed a manager, which we describe by the predicate **isManager**. With the line **Actor** $*\rightarrow*$ $[M]_{[MP]}$: **Cookie**; we ensure that the cookie goes to exactly the person who sent the login (the owner of the pseudonym MP which is – hopefully – the manager). Note that this allows us to faithfully model also the situation where an intruder has found out the password of a manager: in this case he can now obtain also such a cookie (and use this cookie for subsequent impersonation of the manager).

The cookie database is also worth discussing in more detail: we declare

```
nonpublic cookies(agent): agent*cookie set;
```

i.e., similar to the passwords, it is a function parameterized over agent names — in this case the owner of the database. The cookie database simply consists of a set of pairs of agent names and cookies.

We can now re-formulate the **changeGroup** action of the manager to run over a pseudonymous channel, using a previously obtained cookie for authentication:

```
on([?M]_{[?MP]} *\rightarrow* Actor: ?Cookie.(?A->changeGroup(?G1,?G2))
  & cookies(Actor)->contains((?M,?Cookie)) & ?G1->contains(?A)): {
  retract(G1->contains(A));
  G2->contains(A);
}
```

Here the manager is authenticated by the cookie, which is looked up, in conjunction with her name stored in the variable M , in the server's cookie database before granting the transaction. Again this is a faithful model of the real situation: we send a cookie over a TLS channel where the sender is not authenticated — possibly even the pseudonym MP is not the same because a new TLS session had been opened meanwhile. If for some reason the intruder has obtained such a cookie, he can use it to impersonate the manager in such transactions.

This example illustrates how the channel notation can be used to model different levels of granularity of our models: we can either completely abstract from authentication mechanisms and right away use a secure channel, as we did first,

or we can just abstract from TLS but model a credential-based approach like the above password/cookie mechanism. The abstraction has the advantage that we fade out model details and make thus the specification easier to read and work with, while a more detailed specification may allow to model more aspects such as leaking passwords or cookies. Note that again such an intermediate layer could also be addressed with compositional reasoning, i.e., specifying just the credential-based system without concrete applications like `changeGroup` and authentic transmission as a goal.

5 Security Goals

ASLan++ has been geared as a high-level input language for model checking security aspects of distributed systems, and it is therefore crucial to offer a convenient, clear, and expressive way to formalize the desired security properties. The most general way to describe a security property in ASLan++ is to use a first-order temporal-logic formula, defining a set of traces G that satisfy the security properties. An *attack* is then any trace that the system can show and that is not contained in G . The logic that we use is an extension of *LTL* (linear temporal logic); for brevity we refer to it simply as *LTL*. The propositional basis are the ASLan++ facts, and we allow all the standard temporal operators from *LTL* and first-order quantification. The currently available tools however support only fragments of this logic. First, all tools currently only support outermost universal quantification. Second, OFMC and CL-AtSe support only safety properties, i.e. such that every attack manifests itself in a finite trace, while SATMC also supports liveness properties. We do not support any properties that involve multiple traces such as non-interference goals.

To make the specification of simple and common goals as convenient as possible, especially to user without a background in formal logic, we provide several ways to specify goals. In particular we can formulate goals within an entity, allowing to refer to all variables defined in this scope.

Invariants. Goals that should hold during the whole life of an entity instance are stated in the `goals` section of the entity declaration. Properties expected to hold globally during the overall system execution should be given in the outermost entity.

For invariants, basically any *LTL* formula can be given. As an example, consider the meta policy “one cannot have a deputy outside one’s own group” mentioned in [subsection 3.5](#):

```
goals
deputy_in_group: forall A B. [] (B->deputyOf(A) =>
                                (exists G. G->contains(A) & G->contains(B)));
```

where “`[]`” is the “globally” *LTL* operator. However this is outside the supported fragment of all tools (due to the existential quantifier). Taking advantage of the fact that in our model each employee is in exactly one group, which could be specified and checked as a further invariant, we can re-phrase the formula as

```
forall A B G. [] (B->deputyOf(A) => (G->contains(A) => G->contains(B)))
```

As mentioned before, this goal is violated.

Assertions. An assertion is very similar to an invariant, except that it is inserted as a statement in the body of an entity and is expected to hold only at the given point of execution of the current entity instance.

In our example, we can express the expectation that an employee is allowed to access a certain file *F* using a very simple LTL formula:

```
assert can_access_file: Decision = grantedAccess(F);
```

Channel Goals. ASLan++ offers special support for conveniently specifying the usual communication goals like confidentiality, authentication, and the like, called *channel goals*. In our example, in order to state that a manager authenticates to the server on her cookie when sending a `changeGroup` command, we write

```
manager_auth:(_) M *-> S
```

In analogy to the syntax of message transmission, *M* denotes the sender and *S* denotes the receiver of the transmission to which the goal refers. In between them is a symbol for the kind of channel property, in this case “*->” indicating sender authenticity. The goal name `manager_auth` is augmented with a parameter placeholder “(” to indicate that in sub-entities the goal name appears again, in the form of a goal label, with a message term as its argument. Here, the message term is `Cookie`.

This channel goal is stated, as usual, at the level of the `Session` entity and pertains to those message transmissions in which the goal name `manager_auth` re-appears. In our example, we write in the `Manager` entity:

```
[Actor]*->* S: manager_auth:(Cookie).
(e1->changeGroup(g1,g2));
```

and in the `Server` entity:

```
[?M]_ [?MP] *->* Actor: manager_auth:(?Cookie).
(?A->changeGroup(?G1,?G2))
```

The operational semantics of this goal is that whenever the server receives `?Cookie.(?A->changeGroup(?G1,?G2))` from any manager `?M`, the agent denoted by *M* must have sent to the server the `changeGroup` command with the same cookie value (as long as *M* is not the intruder legitimately playing the manager’s role).

It is important to note that *M*’s value is determined dynamically here, depending on the cookie just received. The side condition

```
cookies(Actor)->contains((?M,?Cookie)) % actually learns M here!
```

models that the server looks up the name *M* in its database: it is the name that was stored along with the cookie when the manager logged in. So the manager to be authenticated is not determined statically by the initial value of the parameter *M* of the `Server`.

Secrecy Goals. Very similarly to the channel goal just described, we state

```
shared_secret:(_) {M,S}
```

in the **Session** entity. Its interpretation is that the values annotated by the respective goal labels in the sub-entities must be known only to the agents **M** and **S**. In this case, the confidential value is the password that the manager uses for logging in to the server. Therefore, we write in the **Manager** entity:

```
[Actor]*->* S: login(Actor,shared_secret:(password(Actor,S)));
```

and in the **Server** entity:

```
on([?M]_[?MP] *->* Actor: login(?M, % actually learns M here!
shared_secret:(password(?M,Actor)))
```

The operational semantics is that after the manager or the server has processed the value given as argument of the goal label `shared_secret:(...)`, this value must never show up in the knowledge of the intruder (as long as the intruder does not legitimately play the role described by any of the two entities).

Confidential transmission of a value between two parties can also be stated as a channel goal, but the secrecy goal is more general: it may be used to state that the value is shared between more than two parties, and the confidentiality is meant to be persistent (unless the secrecy goal is retracted or modified dynamically).

6 Conclusion

We have illustrated by means of an example how the major security-relevant features of modern service-oriented architectures can be specified in ASLan++, in particular how to formulate non-linear and inter-dependent workflows, as well as policies and related goals. We have shown how to selectively abstract from communication aspects using secure channels, as well as pseudonymous channels and password- or cookie-based authentication mechanisms. The specifications are clear and readable for web service designers, and at the same time are on a reasonable abstraction level to be feasible for automated verification tools such as those of AVANTSSAR. For instance, the violation of the `deputy_in_group` goal in our example is found by the CL-AtSe back-end in less than a second.

AVANTSSAR Case Studies and Tool Availability. While the running example has been deliberately kept small and simple for presentation, this paper reflects also our experiences in the AVANTSSAR project with real-world case studies [3] from the areas of e-Government, e-Health, and e-Business. There, ASLan++ similarly allows us to have well-structured, easy-to-read descriptions of complex systems that can be effectively analyzed with the automated verification tools of the AVANTSSAR platform within reasonable time (usually much less than 1 hour CPU time). Both the case studies and the AVANTSSAR Tool, including a convenient web interface, are available at www.avantssar.eu.

Related Work. There are a number of specification languages that have similar or overlapping aims. The closest ones are the high-level protocol specification language HLPSSL [10] and the low-level language IF of the predecessor project AVISPA [5]. In fact, experience with these languages had much influence on the ASLan++ design. The most crucial extensions w.r.t. HLPSSL and IF are the integration of Horn clauses and the notion of channels. Moreover, ASLan++ is closer to a programming language than the more logic-oriented HLPSSL and the more low-level description of transition rules of IF. ASLan++ is automatically translated to the more low-level language ASLan, which is an appropriate extension of IF and serves as the input language of the model-checking tools.

In the area of policy specification languages, we are closest to SecPAL [6] and DKAL [18] with their Horn clause specification style. Their relation with a transition system. Another, conceptually quite different approach is KLAIM [14], which allows for specifying mobile processes where access control is formalized using a capability-based type system. Despite the differences, the combination of a policy aspect and a dynamic distributed system bears similar ideas and we plan to investigate as part of future work whether the concepts of the two languages could be connected.

A language focussing on the vertical architecture especially in web services is Capito [17], this is however again built on relatively simple authentication protocols and is not related to the required compositionality results such as [20][11].

One of the pioneering verification frameworks for web services is the Tula-Fale project [7], which in particular supports a convenient way to deal with the details of the message formats such as SOAP. It is based on the verification tool ProVerif [8] using abstraction methods, which represent the entire protocol as a set of Horn clauses. A limitation of this approach is the monotonicity of the abstraction, which forbids for instance to model revocation of access rights. One of our works aims to overcome this limitation while preserving the advantages of abstract interpretation, namely the set-based abstraction approach [19]. It is part of our future work to build a bridge from ASLan++ to that framework. This bridge will consist not only in a translator from ASLan++ to a suitable input language, but also of a mechanism to choose and refine appropriate abstractions.

Acknowledgments. The work presented in this paper was supported by the FP7-ICT-2007-1 Project no. 216471, “AVANTSSAR: Automated Validation of Trust and Security of Service-oriented Architectures”. We thank the anonymous reviewers for their helpful comments.

References

1. Armando, A., Basin, D. A., Boichut, Y., Chevalier, Y., Compagna, L., Cuellar, J., Drielsma, P.H., Heám, P.-C., Kouchnarenko, O., Mantovani, J., Mödersheim, S., von Oheimb, D., Rusinowitch, M., Santiago, J., Turuani, M., Viganò, L., Vigneron, L.: The AVISPA tool for the automated validation of internet security protocols and applications. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 281–285. Springer, Heidelberg (2005)

2. Armando, A., Carbone, R., Compagna, L.: LTL Model Checking for Security Protocols. *Journal of Applied Non-Classical Logics*, special issue on Logic and Information Security, 403–429 (2009)
3. AVANTSSAR. Deliverable 5.3: AVANTSSAR Library of validated problem cases (2010), <http://www.avantssar.eu>
4. AVANTSSAR. Deliverable 2.3 (update): ASLan++ specification and tutorial (2011), <http://www.avantssar.eu>
5. AVISPA Project, <http://www.avispa-project.org>
6. Becker, M.Y., Fournet, C., Gordon, A.D.: Security Policy Assertion Language (SecPAL), <http://research.microsoft.com/en-us/projects/SecPAL/>
7. Bhargavan, K., Fournet, C., Gordon, A.D., Pucella, R.: TulaFale: A security tool for web services. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) *FMCO 2003*. LNCS, vol. 3188, pp. 197–222. Springer, Heidelberg (2004)
8. Blanchet, B.: An efficient cryptographic protocol verifier based on prolog rules. In: *Proceedings of CSFW 2001*, pp. 82–96. IEEE Computer Society Press, Los Alamitos (2001)
9. Burrows, M., Abadi, M., Needham, R.: A Logic of Authentication. *ACM Transactions on Computer Systems* 8(1), 18–36 (1990)
10. Chevalier, Y., Compagna, L., Cuéllar, J., Hanks Drielsma, P., Mantovani, J., Mödersheim, S., Vigneron, L.: A High Level Protocol Specification Language for Industrial Security-Sensitive Protocols. In: *Automated Software Engineering. Proc. SAPS 2004 Workshop*, pp. 193–205. Austrian Computer Society (2004)
11. Ciobâca, S., Cortier, V.: Protocol composition for arbitrary primitives. In: *Proceedings of CSF*, pp. 322–336 (2010)
12. Comon-Lundh, H., Cortier, V.: New decidability results for fragments of first-order logic and application to cryptographic protocols. Technical Report LSV-03-3, *Laboratoire Specification and Verification*, ENS de Cachan, France (2003)
13. Cremers, C.: The scyther tool: Verification, falsification, and analysis of security protocols. In: Gupta, A., Malik, S. (eds.) *CAV 2008*. LNCS, vol. 5123, pp. 414–418. Springer, Heidelberg (2008)
14. De Nicola, R., Ferrari, G., Pugliese, R.: KLAIM: a kernel language for agents interaction and mobility. *IEEE TSE* 24(5), 315–330 (1998)
15. Durgin, N., Lincoln, P., Mitchell, J., Scedrov, A.: Undecidability of bounded security protocols. In: *Proceedings of the Workshop on Formal Methods and Security Protocols* (1999)
16. Escobar, S., Meadows, C., Meseguer, J.: Maude-npa: Cryptographic protocol analysis modulo equational properties. In: *FOSAD*, pp. 1–50 (2007)
17. Gao, H., Nielson, F., Nielson, H.R.: Protocol stacks for services. In: *Proc. of the Workshop on Foundations of Computer Security, FCS* (July 2009)
18. Gurevich, Y., Neeman, I.: Distributed-Knowledge Authorization Language (DKAL), <http://research.microsoft.com/~gurevich/DKAL.htm>
19. Mödersheim, S.: Abstraction by Set-Membership—Verifying Security Protocols and Web Services with Databases. In: *Proceedings of 17th CCS*. ACM Press, New York (2010)
20. Mödersheim, S., Viganò, L.: The Open-source Fixed-point Model Checker for Symbolic Analysis of Security Protocols. In: Aldini, A., Barthe, G., Gorrieri, R. (eds.) *Fosad 2007-2008-2009*. LNCS, vol. 5705, pp. 166–194. Springer, Heidelberg (2009)
21. Rusinowitch, M., Turuani, M.: Protocol insecurity with finite number of sessions is NP-complete. In: *CSFW*, p. 174. IEEE Computer Society, Los Alamitos (2001)

ASLan++ Specification Example

```

specification example
channel_model CCM

entity Environment {

  types
  file < text;
  % a group is an agent set
  cookie < text;

  symbols
  login (agent,symmetric_key): message;
  changeGroup (agent,agent set,agent set): message;
  assignDeputy(agent): message;
  requestAccess(file): message;
  grantedAccess(file): message;
  deniedAccess(file): message;

  nonpublic noninvertible password(agent,agent): symmetric_key;
  nonpublic cookies(agent): (agent * cookie) set;
  % used by the Server to store cookies for managers

  centralServer: agent;
  isManager(agent      ): fact;
  canAccess(agent      ,file): fact;
  isOwner (agent set,file): fact;
  deputyOf (agent      ,agent): fact;

  clauses
  accessDirect(A,G,F): A->canAccess(F) :- G->isOwner(F) & G->contains(A);
  accessDeputy(A,B,F): A->canAccess(F) :- A->deputyOf(B) & B->canAccess(F);

  symbols % for the concrete access examples
  m1: agent;
  e1, e2, e3: agent;
  g1, g2: agent set;
  f1, f2: file;

  entity Session (M, S: agent) {

    entity Server(M, Actor: agent) {
    % Exercise for the reader: how to formulate this for a decentralized system?
    % Hint: introduce either an additional argument (representing the P.o.V.) to
    % all policy judgements, or a modality like "Server->knows(A->canAccess(F))".
    symbols
    MP: public_key; % pseudonym of a manager
    A, B: agent;
    G, G1, G2: agent set;
    F: file;
    Cookie: cookie;
    body {
    while(true) {
    select {
    on([?M]_[?MP] **>* Actor: login(?M, % actually learns M here!
    shared_secret:(password(?M,Actor)))

    & ?M->isManager): {
    Cookie := fresh();
    cookies(Actor)->contains((M,Cookie));
    Actor **>* [M]_[MP]: Cookie;
    }
    on([?M]_[?MP] **>* Actor: manager_auth:(?Cookie).(A->changeGroup(?G1,?G2))
    & cookies(Actor)->contains((?M,?Cookie)) % actually learns M here!
    & ?G1->contains(?A)): {
    retract(G1->contains(A));
    G2->contains(A);
    }
    }
    }
  }
}

```

```

    on(?A *->* Actor: assignDeputy(?B) & ?G->contains(?A) & ?G->contains(?B)): {
      B->deputyOf(A);
    }
    on(?A *->* Actor: requestAccess(?F)): {
      if(A->canAccess(F))
        Actor *->* A: grantedAccess(F);
      else
        Actor *->* A: deniedAccess(F);
    }
  }
}
}
}
goals
  deputy_in_group: forall A B G. [(B->deputyOf(A) =>
    (G->contains(A) => G->contains(B)))]
}
entity Manager(Actor, S: agent) {
  symbols
    Cookie: cookie;
  body {
    [Actor]*->* S : login(Actor, shared_secret:(password(Actor, S)));
    S *->[Actor]: ?Cookie;
    [Actor]*->* S : manager_auth:(Cookie).
      (e1->changeGroup(g1, g2));
  }
}
body { % of Session
  iknows(password(i, S)); % intruder knows its own password
  new Server (M, S);
  new Manager(M, S);
}
goals
  shared_secret:(_) {M, S};
  manager_auth :(_) M *-> S;
}
entity Employee(Actor, S: agent) {
  symbols
    F: file;
    G: agent set;
    Decision: message;
  body {
    if(Actor=e1)
      Actor *->* S: assignDeputy(e2);
    % results in a meta policy violation if "e1->changeGroup(g1, g2)" happens later!

    % get any file currently owned by this employee
    if(?G->contains(Actor) & ?G->isOwner(?F)) {
      Actor *->* S : requestAccess(F);
      % before the decision is received, access rights could have changed...
      S *->* Actor: ?Decision;
      assert can_access_file: Decision = grantedAccess(F);
    }
  }
}
}
body { % of Environment
  % for the concrete access examples:
  m1->isManager;
  g1->contains(e1); g1->contains(e2);
  g2->contains(e3);
  g1->isOwner(f1);
  g2->isOwner(f2);

  any M. Session(M, centralServer); % M may be dishonest!
  new Employee(e1, centralServer);
  new Employee(e2, centralServer);
  % new Employee(e3, centralServer);
}
}
}

```


Orchestration under Security Constraints

Yannick Chevalier, Mohamed Anis Mekki, and Michaël Rusinowitch

LORIA & INRIA Nancy Grand Est, France

FirstName.LastName@loria.fr

Abstract. Automatic composition of web services is a challenging task. Many works have considered simplified automata models that abstract away from the structure of messages exchanged by the services. For the domain of secured services (using e.g. digital signing or timestamping) we propose a novel approach to automated composition of services based on their security policies. Given a community of services and a goal service, we reduce the problem of composing the goal from services in the community to a security problem where an intruder should intercept and redirect messages from the service community and a client service till reaching a satisfying state. We have implemented the algorithm in AVANTSSAR Platform [5] and applied the tool to several case studies.

1 Introduction

To meet frequently changing requirements and business needs, for instance in a federation of enterprises, components are replaced by *services* that are distributed over the network (e.g. the Internet) and *composed* in a demand-driven and flexible way. Service-oriented architectures (SOAs) have gained much attention as a unifying technical architecture that can address the challenges of this ever-evolving environment.

Secured Services. Since it is not acceptable in many cases to grant access to a service to any person present on the Internet, one has to regulate the use of services by *policies*. These policies express the context, including the requester's identity, her credentials, the link between the service and the requester, and higher-level business rules to which a service is subject. They also dictate how the information transmitted between services has to be protected on the wire. In the following we call *secured service* a service that is protected by a security policy.

Composition of Secured Services. Each service may rely on the existence and availability of other (possibly dynamically retrieved) partner services to perform its computation. For this one needs dynamic adaptation and explicit combination of applicable policies, which determine the actions to be executed and the messages to be exchanged in order to satisfy the client requests in accordance with the partners policies. For example, a service granting the access to a resource of a business partner may use a local authentication service, trusted by both partners, to assess the identity of a client and rely on authorization services on both ends that combine their policies to decide whether to grant the access or not.

Contribution of this work. For the domain of secured services we propose a novel approach to automated composition of services based on their security policies. Given a

community of services and a goal service, we reduce the problem of composing the goal from services in the community to a security problem where an intruder should intercept and redirect messages from both the community services and the client in such a way that the client service reaches its final state, defined as an insecure one. The approach amounts to collecting the constraints on messages, parameters and control flow from the components services and the goal service requirements. A constraint solver checks the feasibility of the composition, possibly adapting the message structure while preserving the semantics, and displays the service composition as a message sequence chart. The resulting composed service can be verified automatically for ensuring that it cannot be subject to active attacks from intruders. The services that are input to our system are provided in a declarative way using ASLan [2], a high level specification language. The approach is fully automatic and we show on a case-study how it succeeds in deriving a composed service that is currently proposed as a product by a company.

Paper organization. In Section 3 we explain our approach on a simple example. In Section 4 we introduce the formal model we consider for secured Web services. We explain the mediator synthesis problem in Subsection 4.1, how to represent messages in Subsection 4.2, and services in Subsection 4.3. The composition problem is formally stated in Subsection 4.4 and solved in Subsections 4.5 and 4.6. In Subsection 4.7 we formalize the obtained composed service in the ASLan language in order to verify it automatically against classical security properties. In Section 5 we describe the experiments we have performed on three case-studies and put the focus on one provided by *OpenTrust* company. We show how we can derive automatically a Digital Contract Signing service from their components services. Finally we also prove automatically that the resulting service cannot be subject to active attacks. We conclude in Section 6 where we give several perspectives.

2 Related Work

Most works on Web service composition are based on automata representations of web services [9,11,24], trying to synthesize, from available components, an automata whose behavior simulates the specification of the goal service. These approaches are focusing on control flow. For instance, the Roman model [8] focuses on deterministic atomic actions, and has been extended by data and communication capabilities in the Colombo model [9]. Synthesis of composite services in [9,8] is based on Propositional Dynamic Logic.

Another approach to composition relies on advanced AI planning techniques. For instance [17] applies these techniques at the knowledge level to address the scalability problem, retaining only the features of services that are relevant to compose them. According to [24] most solutions in the literature involve too much human encoding or do not address the problem of data heterogeneity, hence are still far from automatic generation of executable processes. Given the variations in information representation such as message-level heterogeneity *data mediation* is crucial for handling service composition. Hence our objective is to handle (in some cases) structural and semantic heterogeneity as defined by [18]. Furthermore we take into account the effects of the security policy of the services on the format of the messages. An advantage of our approach is that it

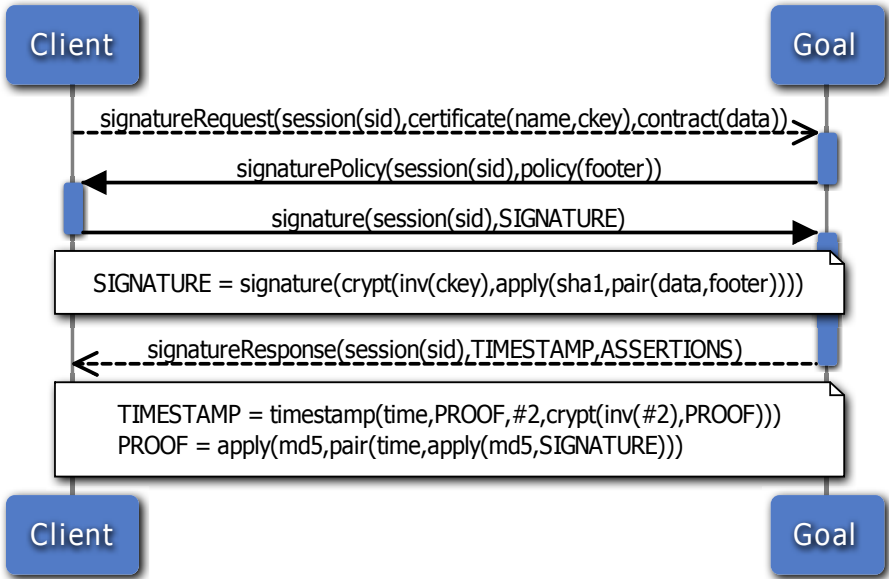


Fig. 1. Time stamping and archiving a digital signature

can handle automatically message structure adaptation since the orchestrator has capabilities (presented by a formal deduction system) to apply operations on messages and build new messages. This provides for free automatic adaptation of messages for proper service communications. We also address the problem of checking that the composed service satisfies some security properties. For the validation of the synthesized service we can employ directly our cryptographic protocol validation tools [6,21].

While our approach focuses on the problems related to message adaptation, it is significantly less expressive than more standard automata-based techniques when considering complex goal services. In particular we consider a bounded orchestration problem in which the number of communications is bounded, thereby excluding iteration completely. As a consequence we believe that the method presented in this paper is more complementary with than a concurrent of these standard methods, and that future work should focus on integrating these approaches in a common framework.

3 Introductory Example

Figure 1 illustrates a composition problem corresponding to the creation of a new service (described here by *Goal*) for appending a timestamp to a digital signature performed by a given partner (described here by *Client*) over some data (described here by *data*) and then submitting it together with the signed data and some other proofs for long time conservation by an archiving third party. More precisely *Goal* should expect a first message from *Client* containing a session identifier *sid*, the *Client*'s certificate

containing his identity and his public key *ckey* and finally the data he wishes to digitally sign. *Goal* should answer with a message containing the same session identifier and a *footer* value to be appended to the data before the *client's* signature. This value aims to capture the fact that the *Client* acknowledges a certain chart (known by *Goal*) before using the service *Goal*. Indeed this is what *Client* is expected to send back to *Goal*. *Goal* should then append to the received digital signature (described by *SIGNATURE*) a timestamp (described by *TIMESTAMP*). The timestamp consists of a *time* value which is bound to the *Client's* signature (through the use of *md5* hash) and signed by a trusted timestamper's private key #2.

Goal should also include a certain number of assertions or proofs about its response message. *ASSERTIONS* is described below and consists of 4 assertions or judgments.

```

ASSERTIONS = ASSRT0, ASSRT1, ASSRT2, ASSRT3
ASSRT0 = assertion(cOCSpr, #0, crypt(inv(#0), cOCSpr))
cOCSpr = ocspr(name, ckey, time)
ASSRT1 = assertion(tsOCSpr, #0, crypt(inv(#0), tsOCSpr))
tsOCSpr = ocspr(#1, #2, time)
ASSRT2 = assertion(arcOCSpr, #0, crypt(inv(#0), arcOCSpr))
arcOCSpr = ocspr(#3, #4, time)
ASSRT3 = assertion(ARCH, #4, crypt(inv(#4), ARCH))
ARCH = archived(session(sid), certificate(name, ckey),
               contract(data), SIGNATURE, TIMESTAMP, ASSRT0, ASSRT1)
#0 in trustedCAKeys
pair(#1, #2) in trustedTSSs
pair(#3, #4) in trustedARs

```

ASSRT0 is a judgment made about the validity of the *Client's* certificate at the time *time* and signed by a certification authority trusted by *Client*. This trust relation is modeled by the fact that the public key of the certification authority is in the set *trusted-CAKeys* representing the public keys of the certification authorities trusted by *Client*. *ASSRT1*, *ASSRT2* represent similar judgments made about the certificates of the used timestamper and archiving service and signed by the same trusted certification authority. On the other hand *ASSRT3* models the fact that the data to be signed by *Client*, its digital signature together with a timestamp and all the proofs obtained for the different involved certificates have been successfully archived by an archiving third party which is in addition trusted by *Client* for this task: here also this trust relationship is modeled by the constraint: *pair(#3, #4) in trustedARs*. These assertions are encoded as certificates embedded in the messages. They represent either a condition ϕ evaluated before accepting a message or a guarantee ψ ensured by the service sending a message.

Finally the use of dashed communication lines in Figure [□](#) refers to additional constraints on the communication channels used by *Client* and *Goal*: in our example this turns to be a transport constraint requiring the use of *SSL*. We can express this constraint in our model by requiring that the concerned messages are ciphered by a symmetric key previously shared between both participants (the key establishment phase is not handled by the composed service).

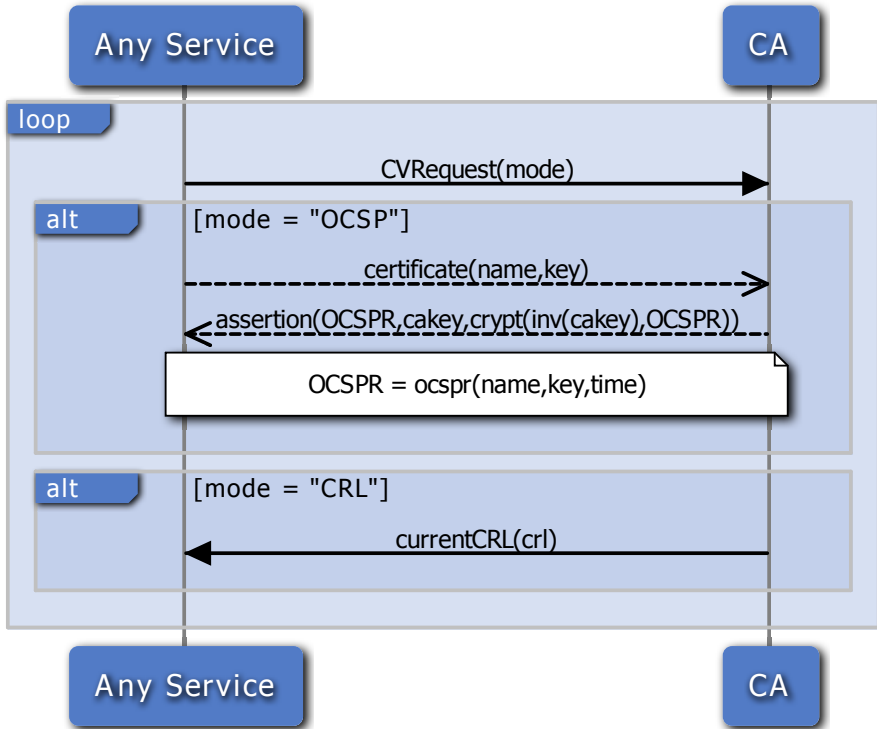


Fig. 2. Available services: Certification Authority

In order to satisfy the requests of *Client*, *Goal* relies on a community of available services ranging from timestampers, and archiving third party to certification authorities.

These services are also given by their interface, i.e. the description of the precise message patterns they accept and they provide in consequence. For instance Figure 2 describes a certification authority *CA* capable of providing two sorts of answers when asked about the validity of a certificate: one is *OCSP*-based (i.e. based on the Online Certificate Status Protocol) and returns a proof containing a real-time time-bound for the validity of a given certificate; while the second only provides the classical Certificate Revocation List *CRL*. Intuitively by inspecting the composition problem one can think that to satisfy the *Client* request the second mode should always be employed with *CA* (provided it is also trusted by the *Client*). One can also deduce that some adaptation should be employed over the *Client*'s messages to obtain the right message patterns (possibly containing assertions) from the community (for example the use of the flag *OCSP* with *CA*).

The solution we propose computes whenever it is possible the sequence of calls to the service community possibly interleaved with adaptations over the already received messages and permitting to satisfy the *Client*'s requests as specified in the composition problem.

4 Formal Description of Service Composition and Adaptation

4.1 Mediator Synthesis

A web service is in standard way described in terms of the interface it presents to the outside world (the possible clients) using the *WSDL* [23] language. This description is structured into ports, each proposing a set of available operations. An operation is then defined by its given in-bound and out-bound message patterns; these patterns are usually described using the *XSD* [26] language and reflects the *XML* message structure. Security constraints can then be defined on top of the service interface description using *WS-SecurityPolicy* [16] annotations. Such annotations can occur at any level in the *WSDL* binding the levels they occur into the security constraints they carry. They range from the service to the message level and typical examples are an *SSL* transport requirement for the whole service or the need to cipher or digitally sign a certain part inside a message pattern (in-bound or out-bound to some operation). We note that the use of *XSD* for the description of message patterns permits the use of the *XPATH* [25] language to write the queries identifying parts inside these message patterns which simplifies the writing of message-level security constraints. We put the focus on SOAP-based (in contrast with RESTful-based) web services. These services rely on the *SOAP* [19] protocol that encapsulates the messages described in the *WSDL* specification of the service. We claim that after (automated) analysis we can collect from the different specification files the descriptions of the different message patterns in-bound and out-bound to all the operations of the service and corresponding to the messages really exchanged by the service (*SOAP* encapsulation included). These descriptions are discussed below.

4.2 Representation of Messages and Security Constraints

We use first-order terms to represent the messages exchanged by a service. We recall this notion below.

Terms. We consider an infinite set of free constants *Consts* and an infinite set of variables \mathcal{X} . For each signature \mathcal{F} (i.e. a set of function symbols with arities), we denote by $T(\mathcal{F})$ (resp. $T(\mathcal{F}, \mathcal{X})$) the set of terms over $\mathcal{F} \cup \text{Consts}$ (resp. $\mathcal{F} \cup \text{Consts} \cup \mathcal{X}$). The former is called the set of ground terms (or messages) over \mathcal{F} , while the latter is simply called the set of terms over \mathcal{F} . Given a term t we denote by $\text{Var}(t)$ the set of variables occurring in t . A substitution σ is an idempotent mapping from \mathcal{X} to $T(\mathcal{F}, \mathcal{X})$ such that $\text{Supp}(\sigma) = \{x \mid \sigma(x) \neq x\}$, the *support* of σ , is a finite set. The application of a substitution σ to a term t (resp. a set of terms E) is denoted $t\sigma$ (resp. $E\sigma$) and is equal to the term t (resp. E) where all variables x have been respectively replaced by the term $x\sigma$. Terms are manipulated by applying *operations* on them. These operations are defined by a subset \mathcal{F}_p of the signature \mathcal{F} called the *set of public symbols*. A context $C[x_1, \dots, x_n]$ is a term in which all symbols are public and such that its nullary symbols are the variables x_1, \dots, x_n . $C[x_1, \dots, x_n]$ is also denoted C when there's no ambiguity and n is called its *length*. We define the *application* of a context C of length n over the sequence of messages $\langle m_1, \dots, m_n \rangle$, denoted by $C \cdot \langle m_1, \dots, m_n \rangle$, to be the image of $C[X_1, \dots, X_n]$ by the substitution $\{X_j \rightarrow m_j\}_{1 \leq j \leq n}$. An *equational theory*

\mathcal{E} is defined by a set E of equations $u = v$ with $u, v \in \mathbb{T}(\mathcal{F}, \mathcal{X})$. We write $s =_{\mathcal{E}} t$ as the congruence relation between two terms s and t . This equational theory is introduced in order to specify the effects of operations on the messages and the properties of messages. We say that a term t is *deducible* from a set of terms E whenever there exists a sequence of elements $\langle t_1, \dots, t_{k_t} \rangle$ in E and a context C_t of length k_t such that $C_t \cdot \langle t_1, \dots, t_{k_t} \rangle =_{\mathcal{E}} t$.

XML Messages. We aim to represent a significant fragment of *XML* messages as described by the *XSD* language using first-order terms defined over a signature given below. The fragment we address corresponds to *XML* elements, described by sequential complex types, i.e. elements having an ordered and a fixed-cardinality set of children. We also abstract away the attributes in *XML* messages. To represent *XML* messages we define the following signature:

$$\mathcal{F} = \left\{ \text{node}_a^n, \text{child}_a^n \mid i \leq a \in \mathbb{N}, n \in \text{Consts} \right\} \cup \left\{ \text{scrypt}, \text{sdcrypt}, \text{crypt}, \text{dcrypt}, \text{sign}, \text{verif}, \text{inv}, \text{invtest}, \top \right\}$$

where the symbol node_a^n represents an *XML* node named n (ranging over the set of constants Consts) and having a children. For each symbol node_a^n we define the set of symbols $\text{child}_1^n, \dots, \text{child}_a^n$ permitting to extract its children. In order to model security constraints holding over exchanged *XML* messages, we also represent the usual cryptographic primitives through the use of symbols: $\text{scrypt}/\text{sdcrypt}$ for symmetric encryption and decryption, $\text{crypt}/\text{dcrypt}$ for asymmetric encryption and decryption, sign/verif for digital signature and its verification, inv to denote key inverses and invtest permitting to test whether a pair of terms $\{t, t'\}$ verifies $t' = \text{inv}(t)$. The constant \top is the result of a successful test. We denote by \mathcal{F}_p , the set of public symbols and assume in the remainder of this chapter that $\mathcal{F}_p = \mathcal{F} \setminus \{\text{inv}\}$.

Some of the symbols represent the possible operations on the messages. Their semantics is defined with the following equational theory:

$$\mathcal{E}_{XML} \left\{ \begin{array}{l} \text{sdcrypt}(\text{scrypt}(x, y), y) = x \quad (D_s) \\ \text{dcrypt}(\text{crypt}(x, y), \text{inv}(y)) = x \quad (D_{as}) \\ \text{verif}(x, \text{sign}(x, \text{inv}(y)), y) = \top \quad (S_v) \\ \text{child}_i^n(\text{node}_a^n(x_1, \dots, x_a)) = x_i \quad (P_{\frac{i}{a}}) \\ \text{invtest}(x, \text{inv}(x)) = \top \quad (I_v) \end{array} \right.$$

We say that a term t is *deducible* (we use also *deduced*) from a set of terms E whenever there exists a sequence of elements $\langle t_1, \dots, t_{k_t} \rangle$ in E and a context C_t of length k_t such that $C_t \cdot \langle t_1, \dots, t_{k_t} \rangle =_{\mathcal{E}_{XML}} t$.

4.3 Representation of Services

We note that the *WSDL* specification of a web service does not precisely list any order of invocation for its operations but only gives their exhaustive list. Moreover this specification does not mention how the input parameters are related to the output parameters for a given operation. The *BPEL* [22] language allows reasoning about such

properties by permitting first to specify a certain workflow logic for the service, and second to specify all the manipulations needed to construct the sent messages given the received ones. In this sense *BPEL* describes *business processes* which are structured workflows of activities ranging over invocation of web service operations, providing of web services operations or manipulation of messages.

We consider services that do not contain any iteration or replication. Here we shall also abstract from internal actions and we shall focus on communications. Therefore a service S will be considered as a sequence of in- and out-bound messages denoted respectively $RCV(m)$ and $SND(m)$.

We assume that all the services we consider are also described in terms of their respective *BPEL* specification and focus only on services described by linear processes, i.e. sequences of activities. Therefore a service S will be considered as a sequence of in- and out-bound messages denoted respectively $RCV(m)$ and $SND(m)$ as described by the following grammar:

$$\begin{array}{ll}
 P, Q := \text{services} & \\
 0 & \text{null service} \\
 RCV(m) \cdot P & \text{input message} \\
 SND(m) \cdot P & \text{output message} \\
 P \parallel Q & \text{AC parallel composition}
 \end{array}$$

Parallel composition of services S_1 and S_2 is denoted by $S_1 \parallel S_2$. It is associative and commutative, and has a unit element 0 , the null process. We consider a community to be a parallel composition of all its available services.

In the following we say that a term t is deducible (or deduced) from a service S whenever t is deducible from the sequence $\langle m_1, \dots, m_{k_S} \rangle$ representing the messages received by the service S .

Transition Semantics. We introduce transition semantics to define how services are executed in interaction with their environment and in particular with clients. The state of a service S can be viewed as the list of remaining operations it has to perform to end properly. For instance the service in state $RCV(r) \cdot S'$ waiting for a message matching r proceeds with $S' \sigma$ if it is added in parallel to a service in state $SND(m) \cdot S$ such that m matches r with some substitution σ . In this case the latter service proceeds then with S . The global configuration is a pair $(\mathcal{S}, \mathcal{E})$ with first component the set of service states, and second component the set of messages that have been sent so far. The evolution of the global configuration is given by the transition rule:

$$(RCV(r) \cdot S \parallel (SND(m) \cdot S' \parallel \dots, \mathcal{E}) \xrightarrow{m} (S\sigma \parallel S' \parallel \dots, \mathcal{E} \cup \{m\})$$

if $\exists \sigma, r\sigma = m$

Such transitions are called *communication transitions*.

The reception of a message instantiates the variables in the receive pattern. This instantiation is applied on the variables remaining in the process that describes the service. A *derivation* is a sequence of transitions. The *size* of derivation is the size of its sequence of transitions. We say that a service has *ended* in a derivation if it is reduced to a null process.

4.4 Web Services Composition Problem

Composition Goal. To answer a client \mathcal{C} request we often need a new service \mathcal{T} to be obtained as a composition of some of the ones that are available in the community. We define the composition goal as the ordered list of messages that \mathcal{C} should receive from \mathcal{T} and that \mathcal{T} should receive from \mathcal{C} . Hence the composition goal is also a service that can be specified with the service grammar given above.

Composition Mediator. We exploit a derivation as follows to generate a mediator. The messages sent by the services are dispatched by the mediator and they can possibly be adapted before assigning them to the proper recipient. In order to express this adaptation capability of the mediator, we simply define the following transition rule:

$$(\mathcal{P}, \mathcal{E}) \xrightarrow{\mathcal{C}} (\mathcal{P}, \mathcal{E} \cup \{m\})$$

if there exists a context C and t_1, \dots, t_n in \mathcal{E} s.t $C[t_1, \dots, t_n] =_{\mathcal{E}_{XML}} m$

We call such transitions *adaptation transitions*.

The problem we are interested in is to check whether a client \mathcal{C} can be satisfied by a composition of services from the community. More formally we can state it as:

Service Composition Problem

- Input:** A community of service $\mathcal{S} = \{S_1, \dots, S_n\}$
 A composition goal \mathcal{C} (specified by the client requests)
- Output:** A derivation from initial state $(\mathcal{S} \cup \{\mathcal{C}\}, \emptyset)$ to a state where \mathcal{C} has ended, and each service in \mathcal{S} has either ended or is in its initial state, if such a derivation exists and the symbol \perp otherwise.

In other word we have to check for the existence of a derivation (applying the transition rules) from an initial state where the client is put in parallel with the community of services and no messages have been sent so far, to a state where all requests from the client have been satisfied (\mathcal{C} has ended) and the services from the community that have been initiated have properly terminated.

4.5 Solving the Composition Problem

Theorem 1. *The Service Composition Problem is NP-complete.*

Sketch of proof: We reduce the Service Composition Problem to showing the existence of an attack on a protocol built from the services and the client (given the \mathcal{E}_{XML} theory). To ensure proper termination of services that are involved in an interaction with the client, we guess at the beginning whether a service S_i will be employed or not. Let $\{S'_1, \dots, S'_m\}$ be the subset of services to be really employed. After this guessing step the composition problem is reduced to the reachability of a configuration $(0, \mathcal{E})$ from a configuration $(\mathcal{C} \parallel S'_1 \parallel \dots \parallel S'_m, \emptyset)$ with $\{S'_1, \dots, S'_m\} \subseteq \{S_1, \dots, S_n\}$

For each service $S \cdot 0$ in $\{\mathcal{C}, S'_1, \dots, S'_m\}$ we introduce a new constant c_S and transform the service $S \cdot 0$ into a service $\overline{S} = S \cdot SND(c_S) \cdot 0$. It is clear that a service S

reduces to the null process if, and only if, \bar{S} sends c_S . Finally we add a monitor service M to the community that checks that all constants are sent. We let

$$M = RCV(cc) \cdot RCV(c_{S'_1}) \dots RCV(c_{S'_m}) \cdot SND(secret) \cdot 0$$

It is clear that M sends $secret$ if and only if all the services C, S'_1, \dots, S'_m reduce to the null process. Thus we have transformed the problem of the reachability of a configuration $(0, \mathcal{E})$ from a configuration $(C \parallel S'_1 \parallel \dots \parallel S'_m, \emptyset)$ into the problem of the reachability of a configuration (P, \mathcal{E}') with $secret \in \mathcal{E}'$ from the initial configuration $(M \parallel C \parallel S'_1 \parallel \dots \parallel S'_m, \emptyset)$. This latter problem is a classic problem for cryptographic protocols and is called the *Protocol insecurity problem*. Since the existence of an attack on a protocol is a problem known to be in NP [20] we can conclude. \square

The protocol insecurity problem corresponding to our composition problem can then be submitted to any state-of-the-art protocol verification tool capable of checking reachability properties. If the composition problem admits a solution we obtain an attack trace (or a conversation trace) describing how the intruder (or the mediator from a composition point of view) succeeded into satisfying the clients requests by applying its adaptation skills on messages exchanged with some services in the community.

For instance Figure 3 illustrates the solution for the composition problem stated in the introductory example.

To fulfill the *Client's* requests (messages $M1$ and $M3$) the mediator first calls (with messages $M4$ and $M5$) the certification authority service denoted by CA to obtain an assertion (message $M6$) stating the validity of the *Client's* certificate. Then he calls the timestamper denoted by TS and trusted by the *Client* (with message $M7$) to obtain a timestamp (message $M8$) and subsequently CA (with messages $M9$ then $M10$) to obtain an assertion (message $M11$) stating the validity of TS 's certificate. Then he calls an archiving third party service ARC trusted by the *Client* (with message $M12$) to obtain assertions (in message $M13$) stating that the *Client's* timestamped signature was correctly archived. Finally the mediator calls CA (with messages $M14$ and $M15$) to obtain the last needed assertion (message $M16$) stating the validity of ARC 's certificate, before successfully answering the last request of the *Client* (with message $M17$).

We remark that the mediator service M is easily extractable from the conversation trace. This can be done by running through all the communication steps in the conversation trace, putting the focus on those involving the mediator and updating its service description as follows:

- if the communication step is $S \rightarrow M : t$, append $RCV(t)$ to M ;
- otherwise, append $SND(t)$ to M .

On the other hand, by definition of a composition problem, a corresponding solution should also describe all the adaptation steps that have to be performed by the mediator. These adaptation steps are abstracted away in the conversation trace depicted in Figure 3, which is a typical result of the state-of-the-art protocol verification tools. For instance, one could intuitively state that the message $M5$ sent by the mediator to CA can be extracted from the message $M1$ (previously sent to him by *Client*) by taking its second child. We present in Section 4.6 an automated procedure permitting to compute all these adaptation steps from a conversation trace similar to the one illustrated in Figure 3.

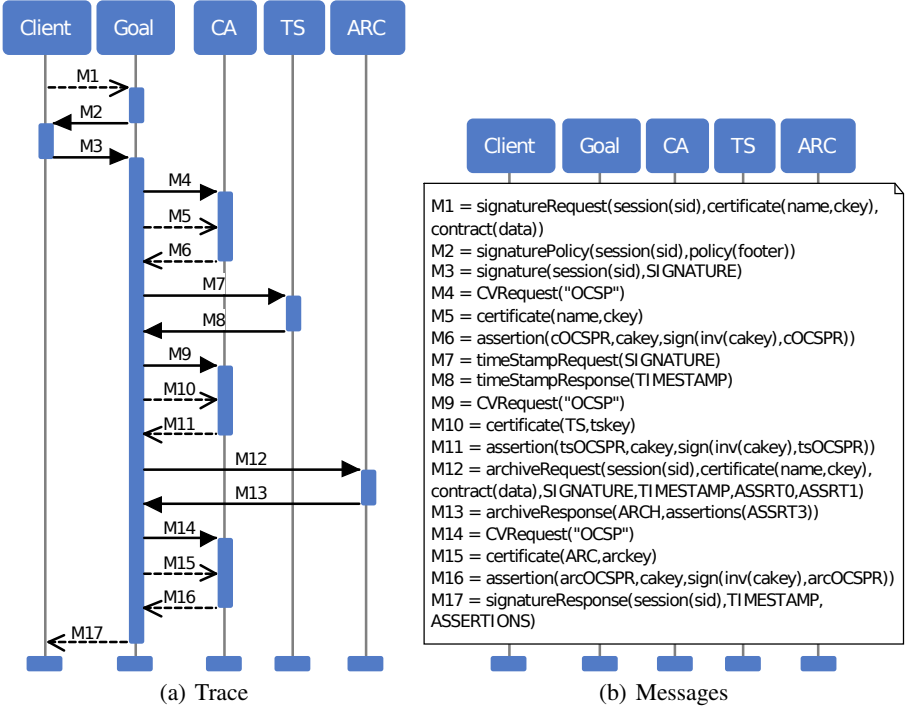


Fig. 3. Solution for the composition problem in the introductory example

4.6 Generating the Mediator's Adaptation Steps

A conversation trace describes partially the solution of a given composition problem. Indeed, it only illustrates the ordered sequence of all communication transitions present in the corresponding derivation. Therefore for all communication step i in the trace, we can extract the following communication transition:

$$\delta_i = (\mathcal{P}_i, \mathcal{E}_i) \xrightarrow{m_i} (\mathcal{P}_{i+1}, \mathcal{E}_i \cup \{m_i\})$$

where $(\mathcal{P}_i, \mathcal{E}_i)$ (resp. m_i) is the configuration before executing step i (resp. the message exchanged at the communication step i). We propose now to enrich this subsequence by interposing the missing adaptation transitions between the existing communication transitions, in order to reconstruct a complete solution of the problem.

Let us first remark that this enrichment may not be unique, and solutions of the composition problem can have an arbitrarily large size. Indeed if the derivation contains at least one adaptation transition, resulting in adding some message m to the environment one can build an infinity of messages (denoted by $T(m)$) by applying public symbols to m and since the environment is finite then one can enrich the derivation with an adaptation transition resulting in adding some new term from $T(m)$ to the environment. To ensure the finiteness of the enrichment we will consider only adaptation transitions that

add subterms occurring in the conversation trace to the environment. Since the inference system associated to the \mathcal{E}_{XML} equational theory enjoys the locality [14] property¹, such an enrichment always exists.

To compute a derivation solving a composition problem given the corresponding conversation trace we first compute for each communication step i , the set \mathcal{E}_i^{new} of the subterms t_1^i, \dots, t_k^i occurring in the trace and deduced by the mediator only starting from that step. We also keep track of \mathcal{C}_i^{new} the sequence of contexts $\langle C_{t_1^i}, \dots, C_{t_k^i} \rangle$ that permit to construct these subterms from the current knowledge in the order according to which they were constructed. Then we construct Δ_i for each communication step i as described below:

$$\Delta_i = \xrightarrow{C_{t_1^i}} (\mathcal{P}_{i+1}, \mathcal{E}_i \cup \{m_i, t_1^i\}) \dots \xrightarrow{C_{t_{k-1}^i}} (\mathcal{P}_{i+1}, \mathcal{E}_i \cup \{m_i, t_1^i, \dots, t_{k-1}^i\}) \xrightarrow{C_{t_k^i}}$$

Finally we construct:

$$\Delta = \delta_1 \Delta_1 \dots \delta_n \Delta_n$$

which is a solution of the composition problem. To prove the last statement, it is sufficient to prove that (i) Δ is a derivation and (ii) Δ solves the composition problem and both are true by construction.

We put the focus now on the computation of the set \mathcal{E}_i^{new} and the sequence \mathcal{C}_i^{new} for all communication step i . We recall that \mathcal{E}_i^{new} is the set of subterms occurring in the trace and deduced by the mediator only starting from the communication step i and that by construction we have: $\mathcal{E}_i^{new} = \mathcal{E}_i \setminus \mathcal{E}_{i-1}$ with $\mathcal{E}_0 = \emptyset$. One important remark here is that only reception steps bring new knowledge to the mediator. After a reception of a message m_i , the mediator tries all the possible applications of equations in \mathcal{E}_{XML} to his current knowledge including the message m_i and possibly computes new subterms occurring in the trace.

We now introduce the notion of *sequents* which we will use to compute the set \mathcal{E}_i for all reception step i .

Definition 1. Given a service S we call γ a sequent of S (denoted by $t_1, \dots, t_k \vdash_f t_0$) an equality $t_0 =_{\mathcal{E}_{XML}} f(t_1, \dots, t_k)$ where f is a public symbol and t_0, \dots, t_k is a sequence of subterms occurring in S . We call respectively t_0 , f and the sequence t_1, \dots, t_k the head, the symbol and the tail of γ and denote them respectively by $h(\gamma)$, $s(\gamma)$ and $t(\gamma)$. We denote the set of all sequents of S by $\Gamma(S)$ and the set of all valid sequents at some step i by $\Gamma_i(S)$.

We say that γ is *valid* at some step i when for all $0 \leq j \leq k$, $t_j \in \mathcal{E}_i$. We remark that if a sequent γ is valid at step i then its head is an element of \mathcal{E}_i . Indeed t_0 is deducible at step i by taking $t_0 =_{\mathcal{E}_{XML}} f(t_1, \dots, t_k)$. We exploit this property to compute the set $\Gamma_i(\text{Mediator})$ for all reception step i of the mediator service.

First we compute $\Gamma(\text{Mediator})$ by running through all the subterms occurring in it and collecting the corresponding sequents. For example a subterm of the form $t = \text{script}(k, m)$ will provide two entries: $k, m \vdash_{\text{script}} t$ and $k, t \vdash_{\text{sdrcrypt}} m$. For each

¹ Informally speaking, this property means that whenever a secrecy attack on a subterm t exists for a given protocol, then the intruder can reproduce a secrecy attack on t where he needs to derive only a subset of the subterms occurring in the protocol or in t .

computed sequent γ we define an integer called its *readiness* and initially set to the size of $t(\gamma)$. This integer is used to compute the validity of a sequent as explained further in this paragraph. For each subterm t we define two fields: *dstep* which will hold the least reception step i where t is deduced and *context* which will hold the context that permitted to deduce t . We also define for each subterm t of s a list of sequents $sequents(t)$ which is initialized by all the sequents γ' such that t appears in the tail of γ' .

Then the idea is to perform a fix-point computation per each step i corresponding to the set $\Gamma_i(Mediator)$. The detailed solution is illustrated by Algorithm 1 which relies on Algorithm 2 and both are given below.

Algorithm 1. Compute Deduced Subterms

Require: $Mediator : Service$

- 1: **for all** $RCV(m_k) \in Mediator$ **do**
 - 2: $deduce(m_k, k)$
 - 3: **end for**
-

We start from subterms that are trivially deduced at some given step *i.e.* all the received messages clearly deduced at their corresponding reception step and try to deduce the new ones by checking whether there exists some sequents having their tails made only of already deduced subterms. In order to select these sequents we make use of the *readiness* field attached to each sequent which is decremented each time one element in its tail is deduced (Algorithm 2, line 4). Since the *readiness* field is initialized by the cardinality of its tail thus whenever $\gamma.readiness$ equals zero at some step then the sequent is also valid at that step.

Algorithm 2. deduce

Require: $t : subterm, i : step$

- 1: **if** $t.dstep > i$ **then**
 - 2: $t.dstep \leftarrow i$
 - 3: $t.context = \gamma$
 - 4: **for all** $\gamma \in sequents(t)$ **do**
 - 5: $\gamma.readiness --$
 - 6: **if** $\gamma.readiness = 0$ **then**
 - 7: $\Gamma_i(p).add(\gamma)$
 - 8: $deduce(h(\gamma), i)$
 - 9: **end if**
 - 10: **end for**
 - 11: **end if**
-

Algorithm 1 runs in linear time in the size of mediator service represented as a directed acyclic graph (DAG).

4.7 Generating the Mediator’s ASLan Specification

The ASLan Language

Background. [\[2\]](#) ASLan (*AVANTSSAR Specification Language*) is defined by extending the *Intermediate Format* (IF) [\[7\]](#). IF is an expressive language for specifying security protocols and their properties, based on multiset rewriting. As described in detail in [\[1\]](#), ASLan extends IF with a number of important features so as to express diverse security policies, security goals, communication and intruder models at a suitable abstraction level, and thereby allow for the formal specification and analysis of complex services and service-oriented architectures.

Most notably, ASLan extends IF with support of Horn clauses and LTL formulas. For instance invariants of the system can be defined by a set of (definite) Horn clauses. Horn clauses allow us not only to capture the deduction abilities of the attacker in a natural way, but also, and most importantly, they allow for the incorporation of authorization logics in specifications of services. Moreover, complex security properties can be specified in Linear Temporal Logic. As shown, for instance, in [\[3\]](#), this allows us to express complex security goals that services are expected to meet as well as assumptions on the security offered by the communication channels.

Syntax and Semantics. Here, we recall the main features of ASLan, pointing the reader to [\[2\]](#) for more details on the language.

An ASLan file consists of several sections, among which:

emphSection *Init*s contains one or more initial states of the transition system. A state of a transition system is a set of variable-free facts.

emphSection *Rules* specifies the transitions of the transition system. A *transition* is a rule containing two parts, a left-hand side (LHS) and right-hand side (RHS). The rule can fire in a state whenever its LHS holds in that state. Moreover, a transition can be labeled with a list of existentially quantified variables whose purpose is to introduce new constants representing fresh data (e.g. nonces).

Example 1. Sample transition.

```

1 step sampleTransition(BankAgent) :=
2   state_BankingService(BankAgent, 1) .
3   iknows(request)
4   =>
5   state_BankingService(BankAgent, 2) .
6   iknows(response)

```

where

- `step` is a keyword used to define a new transition;
- `sampleTransition` is a transition name;
- `BankAgent` is a parameter of the transition;
- `state_BankingService(BankAgent,1), iknows(request), state_BankingService(BankAgent,2), iknows(response)` are facts;

² Excerpts from [\[2\]](#).

- `state_BankingService (BankAgent,1).iknows(request)` is the LHS of the transition;
- `state_BankingService (BankAgent,2).iknows(response)` is the RHS of the transition.

This transition represents the behavior of a banking service that receives a request and then reacts by replying with a response and moving to another state. More precisely, the transition can be fired if there exists a value `val` of variable `BankAgent` such that `state_BankingService (val,1)` and `iknows(request)` are in the current state. The result of firing the transition is to replace the fact `state_BankingService (val,1)` by the fact `state_BankingService (val,2)` and add a new fact `iknows(response)`.

Message sending and receiving are specified using `iknows` facts: the `iknows` in the LHS of a transition stands for receiving a message, while in the RHS of a transition it stands for sending a message. The fact `iknows(request)` of Example 1 will not disappear from the current state, because the predicate `iknows` is persistent: once a message is emitted, it becomes a part of the knowledge of the environment (i.e., of the network or of the intruder) and the environment does not “forget” it. If the LHS of a transition holds in the current state, then it is assumed that the knowledge (represented by a set of ground facts) of the corresponding service is enough to build the messages stated in `iknows` in the RHS of the transition.

We use one predicate per service to specify the service states. By convention the predicate name starts with `state_` followed by the service name, e.g. `state_BankingService` from Example 1.

- **Section Goals:** contains security goals that can be defined as attack states (special states of the transition system) or by means of LTL formulae.

Example 2. Sample attack state.

```

1   attack_state stateName (Msg) :=
2       fact1 (Msg) .
3       fact2 (Msg)

```

Here, attack state `stateName` is reached, if there exists a value `val` of variable `Msg` such that `fact1 (val)` and `fact2 (val)` are in the current state of the transition system.

Section HornClauses: contains a finite set of Horn clauses. They can specify, for instance, the authorization logic.

ASLan Generation Procedure. We build a transition system specified in ASLan and representing a prudent implementation of the mediator service M . First we consider a list $A = \langle a_1, \dots, a_n \rangle$ of all the subterms occurring in M deduced by M (at some communication step) such that for all $1 \leq i < j \leq n$, $a_i.dstep \leq a_j.dstep$ and we associate a fresh variable name per each atom in A through a bijective mapping $\sigma^{-1} : a_i \mapsto X_i$. We then create a state fact `state_M` for M with the following profile type: $type(a_1) * \dots * type(a_n) \rightarrow fact$. For each reception $RCV(m)$ in M we generate an ASLan transition τ having only the fact `iknows($\sigma^{-1}(m)$)` in its RHS. We note that $\sigma^{-1}(m)$ is well defined, since every message m received by p is trivially reachable by her. For each emission $SND(m)$ in M we generate an ASLan transition τ having only the fact `iknows($\sigma^{-1}(m)$)` in its LHS. Again we note here that every message m sent by the mediator M has been already deduced by him and thus $\sigma^{-1}(m)$ is well defined.

We introduce the variable renaming functions $\{VName_j\}_{1 \leq j \leq length(c)}$ to distinguish whether a value has been assigned to the variable X_m or not yet in a transition. For each transition labeled by step j we respectively append to its LHS and RHS the facts $state_M(\langle VName_{j-1}(X_i) \rangle_{1 \leq i \leq n})$ and $state_M(\langle VName_j(X_i) \rangle_{1 \leq i \leq n})$ where the functions $VName_j$ map variables to ASLan variable names as follows:

$$VName_j(X_i) = \begin{cases} NI_X_i, & \text{if } \sigma(X_i).dstep \geq j; \\ X_i, & \text{otherwise.} \end{cases}$$

Finally we specify the initial state of the partner: $state_n(\langle ni_X_i \rangle_{1 \leq i \leq n})$. Informally speaking we initialize (with dummy values) the variables corresponding to atoms that will be seen in received messages or generated as nonces in messages to be sent.

5 Experimental Results

5.1 Avantssar Platform

The above mediator construction has been implemented in AVANTSSAR Platform [5], as the Orchestrator module. A solution of the orchestration problem (the description of the mediator) is automatically extracted from the attack trace and then translated to ASLan using the *Trace2ASLan* module. It provides us with an operational implementation of the new feature provided by the composed service (or mediator). The combination of this implementation and the available services is validated with respect to regular security properties (and in prescript of all other partner services) in presence of an active intruder. The validation task is performed by the AVANTSSAR backends. The overall architecture is displayed in Figure 5. We have applied the AVANTSSAR Platform to three case-studies from the AVANTSSAR Test Library [4] (putting the focus on composition problems). In the following we describe one of these case-studies then we provide the execution times needed to solve the three corresponding orchestration problems as well as the one in the introductory example.

5.2 Running Case Study

Description. We have applied the AVANTSSAR Platform to a *digital contract signing* (DCS) case study, provided by OpenTrust. A *Business Portal* (BP) is provided to parties that plan to digitally sign a contract. The goal of this case study is to automatically compose a *security server* (SeS) that will interact with this security portal as well as with available services to satisfy the security constraints. We assume that the community of services available to compose *SeS* contains the following services:

Timestamp: An external service that provides a timestamping functionality. We abstract the protocol employed to communicate with this service with a simple payload exchange with an assertion guaranteeing the timestamp's freshness;

PKI: A *Public Key Infrastructure* (PKI) is employed to check the validation keys of the customers. Again, we abstract away the protocols employed to communicate with this or these PKI, and rely on an assertion to characterize the PKI functionality.

Archiver: An archiver service is also accessible. We are doing coarse grain composition, and simply abstract this service with an assertion stating that this service is trusted for long-term storage.

Security Constraints. There were several constraints on the SeS. We list here the main ones: i) The exchange between the BP and the SeS must be secured using the HTTPS protocol ii) The contracts have to be stored securely.

Client Service. In addition to the compliance with the above constraints, the SeS provider that we generate has to be able to be the *security server partner* in the BP process. To this end we have extracted the message exchange session with the *SeS partner* from the BP definition, and imposed that the generated service interacts with the BP. In the process the customers were completely abstracted away.

Orchestration. We run CL-Atse [21], one of the back-end of Avispa Tool suite, with the ASLan specification described above. It returns an attack on the secrecy of a newly introduced constant (signaling the end of the client service), which corresponds to the trace of the messages exchanged between the component services (and the Business Portal). This trace is displayed in Figure 4 and can be directly translated to a composed service for Digital Contract Signing.

Validation of the Composed Service. The trace computed for the composition and adaptation of services has been automatically translated to an ASLan specification of the mediator which has been added in parallel with the specification modeling the available services and the client. We have applied CL-Atse to verify some security properties of the resulting composition to validate the secrecy of the proof record and the authentication of the BP by the security server.

5.3 Testing Benchmark

We briefly describe two other composition problems from the AVANTSSAR Library.

- *Public Bidding (PB)*: PB illustrates a secure document exchange, and aims at providing a web portal (the *Bidding Portal (BiP)*) to manage an online call for tender, and also Bidders' proposal submissions. The composition problem associated to this case-study amounts to generating the BiP's behavior satisfying the requests of two bidders.
- *Car Registration Process (CRP)*: CRP models an e-government scenario, where a citizen have a secure access point, enabling communication with government officers (with a hierarchical chief, the *Officer Head (OfH)*) and service providers in an easily usable and secure way. From this portal, citizens may access a great variety of services with different authentication, authorization, and protection requirements. The composition problem associated to this case-study amounts to generating the OfH's behavior that leads the government officers to satisfy some citizen's car registration request.

One common concern in these case-studies (including DCS) is to guarantee the legal value of the electronic documents produced. The security requirements imposed on such platforms are highly critical since the probative value of digitally signed documents relies on the conditions under which they have been produced and validated.

Table 1. Execution Times

Case-Study	Composition Time	ASLan Generation Time	Verification Time
Introductory Example	30 s	163 ms	708 ms
DCS	1 m	659 ms	3 m
PB	1 m	4 s	18 m
CRP	2 m	3 s	4 m

Table 1 illustrates per each case-study the different execution times needed to solve the composition problem, to generate the ASLan specification of the mediator (and add it in parallel to the initial specification) and finally to verify the resulting specification.

6 Conclusion

Relying on cryptographic protocols analysis methods we succeeded into solving a class of web services composition problem. The next step is to generate an executable mediator service from the produced ASLan specification. We also need to ensure for security reasons that the generated mediator code controls and protects as much as possible the messages it sends and receives. We have already obtained initial results in this direction. A second research line is to extend the presented works to services with more complex workflows.

References

1. AVANTSSAR. Deliverable 2.1: Requirements for modelling and ASLan v.1 (2008), <http://www.avantssar.eu>
2. AVANTSSAR. Deliverable 2.3: ASLan final version with dynamic service and policy composition (2010), <http://www.avantssar.eu>
3. AVANTSSAR. Deliverable 5.1: Problem cases and their trust and security requirements (2008), <http://www.avantssar.eu>
4. AVANTSSAR. Deliverable 5.4: Assessment of the AVANTSSAR Validation Platform (2010), <http://www.avantssar.eu>
5. AVANTSSAR. AVANTSSAR Platform (2010), <http://www.avantssar.eu>
6. Armando, A., et al.: The Avispa Tool for the automated validation of internet security protocols and applications, <http://www.avispa-project.org/>
7. AVISPA Deliverable 2.3: The Intermediate Format (2003), <http://www.avispa-project.org>
8. Berardi, D., Calvanese, D., De Giacomo, G., Lenzerini, M., Mecella, M.: Automatic Composition of E-services That Export Their Behavior. In: Orłowska, M.E., Weerawarana, S., Papazoglou, M.P., Yang, J. (eds.) ICSOC 2003. LNCS, vol. 2910, pp. 43–58. Springer, Heidelberg (2003)

9. Berardi, D., Calvanese, D., De Giacomo, G., Hull, R., Mecella, M.: Automatic composition of transition-based semantic web services with messaging. In: Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, pp. 613–624. ACM, New York (2005)
10. Bultan, T., Su, J., Fu, X.: Analyzing conversations of Web services. In: Proceedings of the Internet Computing, pp. 18–25. IEEE, Los Alamitos (2006)
11. Bultan, T., Fu, X., Hull, R., Su, J.: Conversation specification: a new approach to design and analysis of e-service composition. In: Proceedings of the International Conference on World Wide Web, WWW 2003, pp. 403–410 (2003)
12. Colombo, M., Di Nitto, E., Mauri, M.: SCENE: A Service Composition Execution Environment Supporting Dynamic Changes Disciplined Through Rules. In: Dan, A., Lamersdorf, W. (eds.) ICSC 2006. LNCS, vol. 4294, pp. 191–202. Springer, Heidelberg (2006)
13. Dolev, D., Yao, A.: On the Security of Public-Key Protocols. *IEEE Transactions on Information Theory* 2(29) (1983)
14. McAllester, D.A.: Automatic Recognition of Tractability in Inference Relations. *Journal of the ACM* 40, 284–303 (1993)
15. Monfroy, E., Perrin, O., Ringeissen, C.: Dynamic Web Services Provisioning with Constraints. In: Meersman, R., Tari, Z. (eds.) OTM 2008, Part I. LNCS, vol. 5331, pp. 26–43. Springer, Heidelberg (2008)
16. Oasis Technical Committee on Secure Exchange. Ws-securitypolicy 1.2 (2007), <http://doc.oasis-open.org/ws-sx/ws-securitypolicy/200702/ws-securitypolicy-1.2-spec-cd-02.pdf>
17. Pistore, M., Marconi, A., Bertoli, P., Traverso, P.: Automated Composition of Web Services by Planning at the Knowledge Level. In: International Joint Conference on Artificial Intelligence, IJCAI (2005)
18. Sheth, A.P., Kashyap, V.: So far (schematically) yet so near (semantically). In: Hsiao, D.K., Neuhold, E.J., Sacks-Davis, R. (eds.) DS-5. IFIP Transactions, vol. A-25, pp. 283–312. North-Holland, Amsterdam (1992)
19. World Wide Web Consortium. Simple Object Access Protocol 1.2 (April 2007), <http://www.w3.org/TR/soap12-part1>
20. Rusinowitch, M., Turuani, M.: Protocol insecurity with finite number of sessions is NP-complete. In: Proc. 14th IEEE Computer Security Foundations Workshop, Cape Breton, Nova Scotia (June 2001)
21. Turuani, M.: The CL-Atse Protocol Analyser. In: Pfenning, F. (ed.) RTA 2006. LNCS, vol. 4098, pp. 277–286. Springer, Heidelberg (2006)
22. Oasis Consortium. Web Services Business Process Execution Language Version 2.0. (January 23, 2006), http://www.oasis-open.org/committees/documents.php?wg_abbrev=wsbpel
23. World Wide Web Consortium. Web Services Description Language (WSDL) 1.1 (March 15, 2001), <http://www.w3.org/TR/wsdl>
24. Wu, Z., Gomadam, K., Ranabahu, A., Sheth, A., Miller, J.: Automatic Composition of Semantic Web Services Using Process Mediation. In: ICEIS, vol. (4), pp. 453–462 (2007)
25. World Wide Web Consortium. XML Path Language (XPath) 2.0. (January 23, 2007) <http://www.w3.org/TR/xpath20/>
26. World Wide Web Consortium. XML Schema Definition (XSD) (March 2005), <http://www.w3.org/XML/Schema>

7 Appendix

In Fig. 4 we have employed the following abbreviations:

$$M_1 = \{signature_sent(s1 \cdot contract \cdot \{sha1(contract \cdot s1 \cdot signaturepolicy)\}_inv(pk s1)) \cdot \{sha1(contract \cdot s1 \cdot signaturepolicy)\}_-(inv(pk s1))\}_kbpss$$

$$M_2 = timestamp_ok(sha1(contract \cdot s1 \cdot signaturepolicy) \cdot n267(N) \cdot sha1(sha1(contract \cdot s1 \cdot signaturepolicy) \cdot n267(N))_-(inv(pkts))) \cdot n267(N) \cdot \{sha1(sha1(contract \cdot s1 \cdot signaturepolicy) \cdot n267(N))\}_-(inv(pkts))$$

$$M_3 = timestamp_ok(sha1(contract \cdot s1 \cdot signaturepolicy) \cdot n267(N) \cdot \{sha1(sha1(contract \cdot s1 \cdot signaturepolicy) \cdot n267(N))\}_inv(pkts)) \cdot timestamp_ok(sha1(contract \cdot s2 \cdot signaturepolicy) \cdot n258(N) \cdot \{sha1(sha1(contract \cdot s2 \cdot signaturepolicy) \cdot n258(N))\}_inv(pkts)) \cdot n267(N) \cdot \{sha1(sha1(contract \cdot s1 \cdot signaturepolicy) \cdot n267(N))\}_inv(pkts) \cdot n258(N) \cdot \{sha1(sha1(contract \cdot s2 \cdot signaturepolicy) \cdot n258(N))\}_inv(pkts))$$

$$M_4 = signature_sent(s1 \cdot contract \cdot \{sha1(contract \cdot s1 \cdot signaturepolicy)\}_inv(pk s1)) \cdot signature_sent(s2 \cdot contract \cdot \{sha1(contract \cdot s2 \cdot signaturepolicy)\}_inv(pk s2)) \cdot \{sha1(contract \cdot s1 \cdot signaturepolicy)\}_inv(pk s1) \cdot \{sha1(contract \cdot s2 \cdot signaturepolicy)\}_inv(pk s2)$$

$$M_5 = \{contract \cdot s1 \cdot signaturepolicy \cdot \{sha1(contract \cdot s1 \cdot signaturepolicy)\}_-(inv(pk s1)) \cdot n267(N) \cdot \{sha1(sha1(contract \cdot s1 \cdot signaturepolicy) \cdot n267(N))\}_-(inv(pkts)) \cdot \{sha1(s1 \cdot pk s1)\}_inv(pk pki) \cdot \{sha1(contract \cdot s2 \cdot signaturepolicy)\}_-(inv(pk s2)) \cdot n258(N) \cdot \{sha1(sha1(contract \cdot s2 \cdot signaturepolicy) \cdot n258(N))\}_inv(pkts) \cdot \{sha1(s2 \cdot pk s2)\}_inv(pk pki)\}_kssarc$$

$$M_6 = archive_ok(contract \cdot s1 \cdot signaturepolicy \cdot \{sha1(contract \cdot s1 \cdot SignaturePolicy(3))\}_inv(pk s1) \cdot n267(N) \cdot \{sha1(sha1(contract \cdot s1 \cdot signaturepolicy) \cdot n267(N))\}_-(inv(pkts)) \cdot \{sha1(s1 \cdot pk s1)\}_inv(pk pki) \cdot \{sha1(contract \cdot s2 \cdot signaturepolicy)\}_inv(pk s2) \cdot n258(N) \cdot \{sha1(sha1(contract \cdot s2 \cdot signaturepolicy) \cdot n258(N))\}_inv(pkts) \cdot \{sha1(s2 \cdot pk s2)\}_inv(pk pki))$$

$$M_7 = archive_ok(contract \cdot s1 \cdot signaturepolicy \cdot \{sha1(contract \cdot s1 \cdot signaturepolicy)\}_inv(pk s1) \cdot n267(N) \cdot \{sha1(sha1(contract \cdot s1 \cdot signaturepolicy) \cdot n267(N))\}_inv(pkts)) \cdot \{sha1(s1 \cdot pk s1)\}_inv(pk pki) \cdot \{sha1(contract \cdot s2 \cdot signaturepolicy)\}_inv(pk s2) \cdot n258(N) \cdot \{sha1(contract \cdot s2 \cdot signaturepolicy \cdot n258(N))\}_inv(pkts) \cdot \{sha1(s2 \cdot pk s2)\}_-(inv(pk pki)))$$

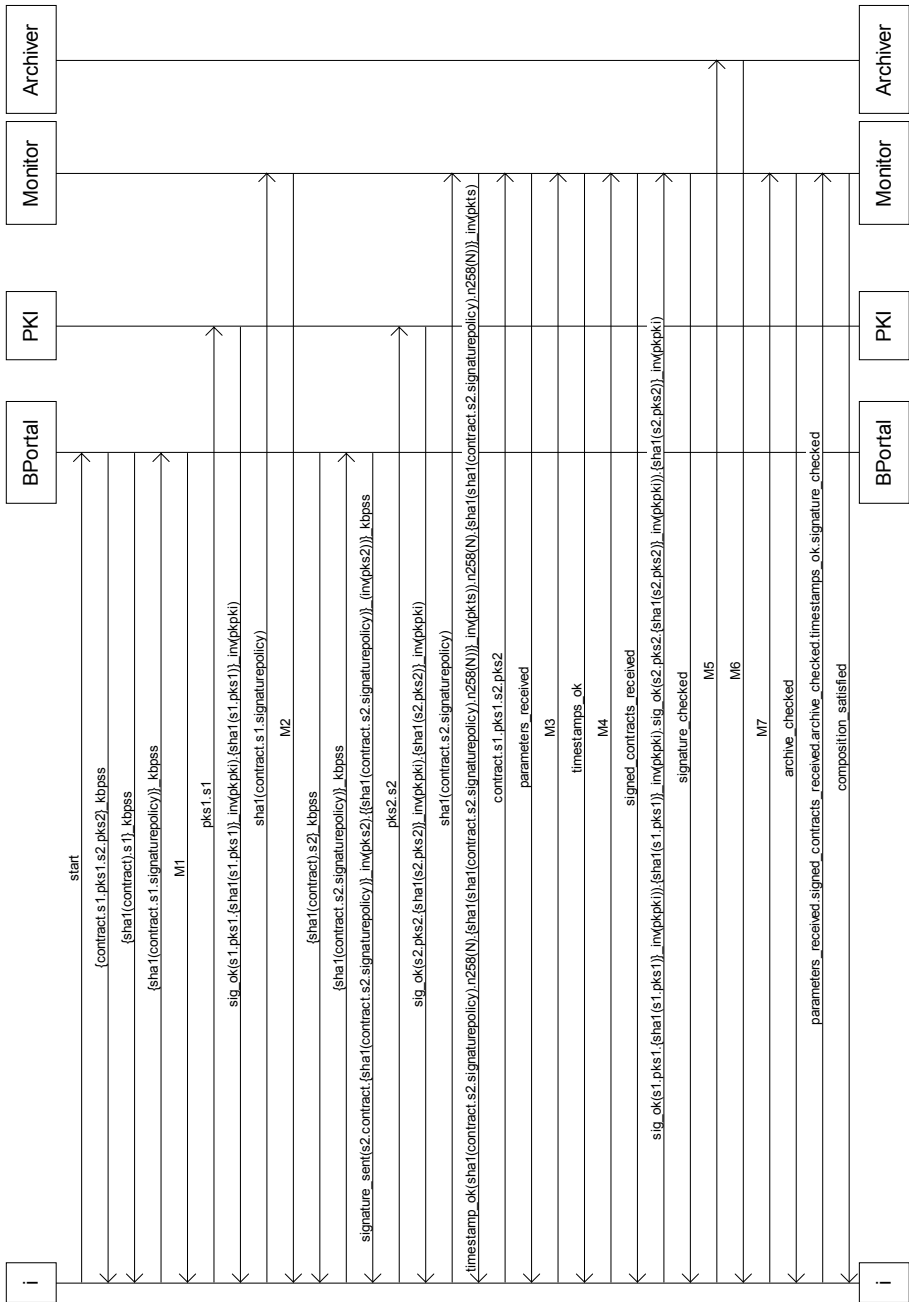
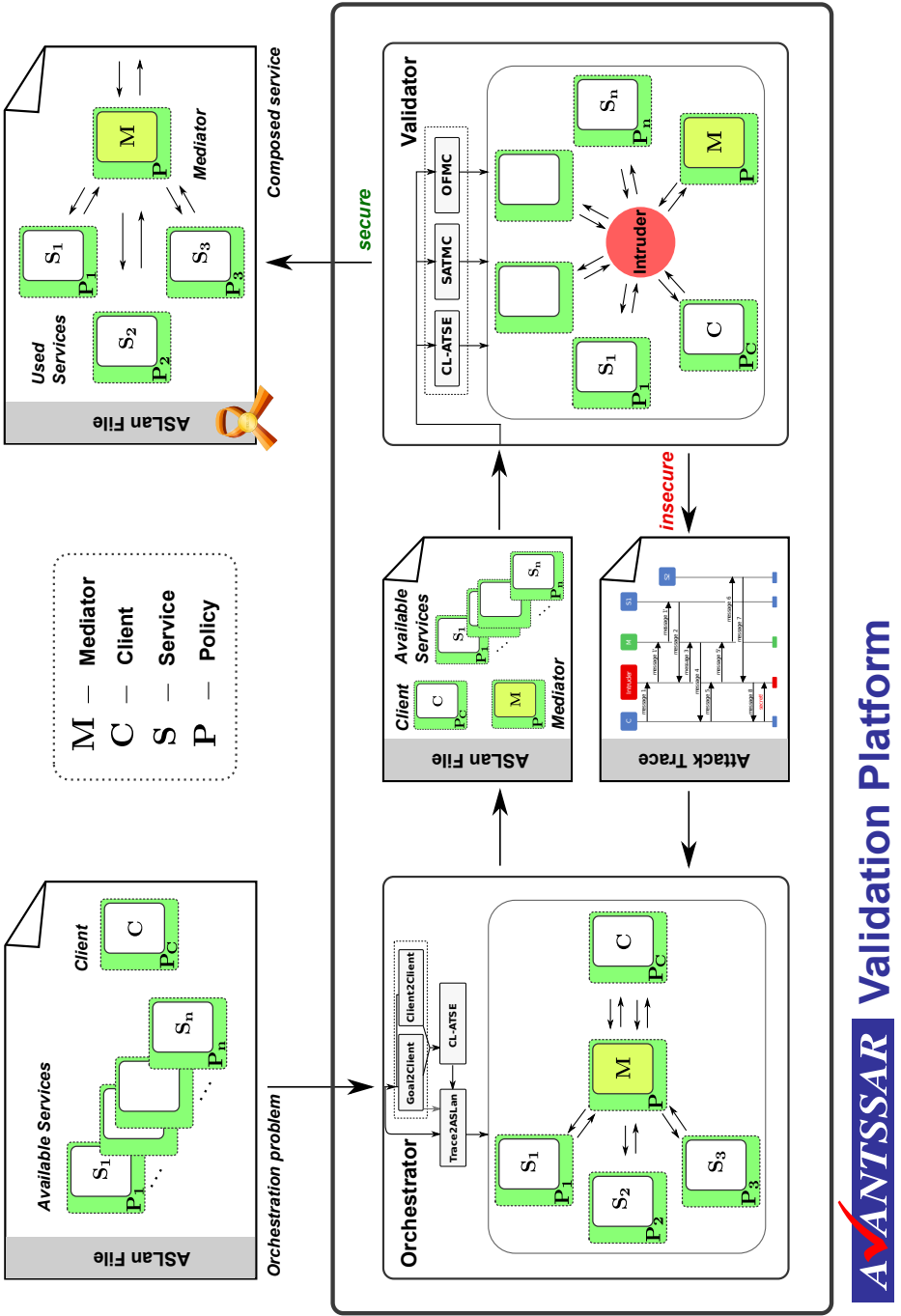


Fig. 4. Sequence Diagram for Digital Contract Signing (the intruder i stands for the security server)



AVANTSSAR Validation Platform

Fig. 5. Avantssar Platform

Customizing Protocol Specifications for Detecting Resource Exhaustion and Guessing Attacks*

Bogdan Groza and Marius Minea

Politehnica University of Timișoara and Institute e-Austria Timișoara
bogdan.groza@aut.upt.ro, marius@cs.upt.ro

Abstract. Model checkers for security protocols often focus on basic properties, such as confidentiality or authentication, using a standard model of the Dolev-Yao intruder. In this paper, we explore how to model other attacks, notably guessing of secrets and denial of service by resource exhaustion, using the AVANTSSAR platform with its modelling language ASLan. We do this by adding custom intruder deduction rules and augmenting protocol transitions with constructs that keep track of these attacks. We compare several modelling variants and find that writing deductions in ASLan as Horn clauses rather than transitions using rewriting rules is crucial for verification performance. Providing automated tool support for these attacks is important since they are often neglected by protocol designers and open up other attack possibilities.

1 Introduction and Motivation

Formal verification tools provide an efficient means for automatic verification of security protocols, once models of these have been written, e.g., some variant of symbolic transition systems. Usually, the focus is on verification of standard security goals, such as authenticity and confidentiality. However, in many cases, satisfying these goals is not sufficient to consider a protocol safe and a more in-depth analysis to rule out other kinds of attacks is necessary.

This paper focuses on two such attacks which are not handled routinely by many protocol verifiers, namely guessing attacks and denial of service (DoS). Both of these attacks are a main concern in protocol design. Guessing attacks are relevant because users tend to choose weak passwords, and some values such as PIN codes have intrinsically low entropy. They can become the weakest link in more complex protocols, leading to other attacks as well. Resource exhaustion is relevant as a common source of DoS as well as from an economic point of view if one considers ruling out protocol designs that can be exploited to make honest participants spend unreasonable amounts of resources, time or memory.

* This work is supported in part by FP7-ICT-2007-1 project 216471, AVANTSSAR: Automated Validation of Trust and Security of Service-oriented Architectures and by strategic grant POSDRU/21/1.5/G/13798 of the Human Resources Development Programme 2007-2013, co-financed by the European Social Fund – Invest in People.

Our research is performed in the framework of the AVANTSSAR project, where security protocols and services are specified as transition systems in the ASLan specification language, based on set rewriting. ASLan models can be analyzed by three different model checkers as back-ends: CL-Atse [19], OFMC [4] and SATMC [2]. None of them handles DoS or guessing attacks by default.

We present several ways to model custom deduction rules in ASLan. The purpose is to devise a way to augment protocol models so they can be analyzed for guessing and DoS attacks without changing the model checkers.

To introduce customized attack rules at the ASLan level there are two main options: adding new intruder transitions and/or adding Horn clauses. The first approach results in mixing protocol steps with the new customized transitions, which can significantly increase state space explosion and verification time. Using Horn clauses is more efficient but faces several issues. both due to the ASLan design (Horn clauses cannot add to intruder knowledge, which would be needed in guessing) and because the model checkers differ in their level of support for Horn clauses and in their search strategies. Consequently, to express customized attack rules, we need to write additional transition rules as well as Horn clauses.

Section 2 presents the ASLan specification language focusing on the main features relevant for our modelling. In Section 3, we briefly present the principles behind our analysis of DoS attacks and then details of their modelling with customized transitions. Section 4 describes the modelling of guessing attacks, with several versions using customized transitions and, for efficiency, Horn clauses for intruder deductions. We conclude in Section 5 with a discussion of the results.

2 The ASLan Specification Language

The AVANTSSAR specification language ASLan is an expressive language for specifying security protocols and services as well as their policies, based on set rewriting. In the following, we give a simplified account of the language, focusing on the features which are most relevant for our customized modelling of attack rules. A full description of the language is given in [3].

ASLan models are transition systems in which each state is modeled by a set of ground facts. Predefined types include `message` and its subtypes (`agent`, `private_key`, `public_key`, `symmetric_key`, `text`). The user can declare additional type symbols, functions and facts (predicates) with their type signatures.

For example, consider the MS-CHAP protocol, a known target for guessing attacks. Figure 1 presents the description in Alice-and-Bob notation, together with an ASLan transition rule for role A, who on receiving nonce N_B in step 2 responds with N_A and a hash computation in step 3.

Here, `state_A` is a fact that tracks the state of principal A, including an instance identifier `ID`, a step counter that changes from 1 to 2, and other known values, including the identity of B, the shared key, the hash function, and the two nonces (which become known as a result of the step). Communication is modeled using the fact `iknows` (on the left-hand side for receive, and on the right-hand side for send), since anything transmitted becomes part of the intruder knowledge. Conjunction of facts is represented by a dot; `apply` represents function

application and `pair` message concatenation. The `exists` keyword specifies the creation of a fresh value as part of the transition.

<ol style="list-style-type: none"> 1. $A \rightarrow B : A$ 2. $B \rightarrow A : N_B$ 3. $A \rightarrow B : N_A, H(k_{AB}, N_A, N_B, A)$ 4. $B \rightarrow A : H(k_{AB}, N_A)$ 	<pre> step step_1(A,B,H,ID,Kab,Na,Nb,Na0,Nb0) := state_A(A,ID,1,B,Kab,H,Na0,Nb0) .iknows(Nb) =[exists Na]=> state_A(A,ID,2,B,Kab,H,Na,Nb) .iknows(pair(Na,apply(H, pair(Kab,pair(Na,pair(Nb,A)))))) </pre>
---	---

Fig. 1. MS-CHAP v2 protocol and ASLan transition rule

Let \mathcal{F} be the set of ground facts; the set of all possible states is then $\mathcal{S} = 2^{\mathcal{F}}$. An ASLan model defines a transition system $M = \langle \mathcal{S}, \mathcal{I}, \rightarrow \rangle$, where $\mathcal{I} \subseteq \mathcal{S}$ is the set of initial states and $\rightarrow \subseteq \mathcal{S} \times \mathcal{S}$ is the transition relation.

In an ASLan model, the set of initial states is a conjunction of facts. Transitions are rewrite rules where both sides are conjunctions of facts. A transition can be taken from any state that contains the facts on the left-hand side; these are removed from the state and replaced by the facts on the right-hand side. As an exception, `iknows` (intruder knowledge) is a persistent fact and does not disappear, even if written on the left-hand side and being omitted on the right.

Formally, we first define the closure $\lceil S \rceil^H$ of a state S with respect to the set H of Horn clauses in the model as the set of all ground facts that can be derived from S using H . More precisely, $\lceil S \rceil^H$ is the smallest set containing S such that $\forall F \leftarrow F_1, \dots, F_n \in H, \forall \sigma. \bigcup_{1 \leq i \leq n} F_i \sigma \subseteq \lceil S \rceil^H \Rightarrow F \sigma \in \lceil S \rceil^H$ where σ is any substitution function that maps the variables of the Horn clause $F \leftarrow F_1, \dots, F_n$ to ground terms.

A transition rule in ASLan has the form $PF.NF \Rightarrow[V] R$, where PF is a set of positive facts, NF is a set of negative (negated) facts, V is a set of fresh introduced variables, and the right-hand side R is a conjunction of facts.

We can now define the transition relation \rightarrow as follows: there is a transition $S \rightarrow S'$ iff there exists a transition rule $PF.NF \Rightarrow[V] R$ and a substitution σ from the variables of PF to ground terms such that following conditions hold:

- $PF\sigma \subseteq \lceil S \rceil^H$, i.e., the positive facts on the left-hand side hold in $\lceil S \rceil^H$
- $NF\sigma\sigma' \cap \lceil S \rceil^H = \emptyset$ for all substitutions σ' such that $NF\sigma\sigma'$ is ground, i.e., the negative facts cannot hold in $\lceil S \rceil^H$
- $S' = (S \setminus PF\sigma) \cup R\sigma\sigma''$, where σ'' is any substitution such that for all $v \in V$, $v\sigma''$ does not occur in S (i.e., variables in V are substituted with fresh terms).

The combination of transition rules and Horn clauses in the language implies the existence of two kinds of facts. *Explicit* facts are introduced by the right-hand side of transition rules and are persistent unless removed by a later transition (if present on the left-hand side but not the right-hand side). *Implicit* facts are introduced by Horn clauses and are recomputed as part of the state closure after each transition step. To ensure a consistent semantics, explicit facts (including

the intruder knowledge (`iknows`) cannot appear in the conclusion of a Horn clause. This impacts our design of guessing rules, which must add intruder knowledge.

These definitions lead to an execution model for an ASLan specification that alternates Horn clause deductions and transition steps: first, the set of facts comprising a state is augmented by the facts obtained by performing the transitive closure of the Horn clauses, and then one of the applicable transition rules is chosen and executed, after which the entire process is repeated. In particular, this makes Horn clauses suitable for modelling intruder deduction and any additional processing necessary for attack detection, as Horn clause deductions are performed after each transition step.

3 Customized Transitions for Detection of DoS Attacks by Resource Exhaustion

We formalize costs and attack conditions in order to detect DoS attacks by resource exhaustion. While we focus on computation resources due to the varying cost of cryptographic primitives, costs could be associated to memory consumption or other resources as well.

Resource exhaustion DoS attacks can be divided according to the behaviour of the adversary in two categories: one is *abusive* use of the service by clients which willingly or not deplete the server from resources, the other is *malicious* use in which adversaries manipulate protocol messages and make honest principals waste computational time without reaching protocol goals. For the first case, we consider an attack feasible if the initiator can force repeated use of the protocol, which leads to resource depletion. For the second case we consider the protocol under attack when principals reach states in which their beliefs about the protocol are wrong, e.g., messages are accepted from impersonated senders. Cutting down communication is not an issue since the intruder can do this for any protocol and protocol design cannot give countermeasures to it. In both cases, to deem a resource exhaustion attack successful we must evaluate costs for both the adversary and honest principals. An attack is flagged as successful when both the cost of the adversary is lower and one of the two situations hold: the adversary is the initiator or the principal's beliefs are wrong.

3.1 Defining Costs and Augmenting Transitions

Costs can be treated according to the framework of Meadows [16], which uses a monoid structure; this approach is also used in follow-up related work [17,18]. The cost set employed is $S = \{0, low, medium, high\}$, and the sum of two costs is simply defined as their maximum: $\forall a, b \in S, a + b = \max(a, b)$. This can be easily modeled in ASLan by using a fact for summing costs, as shown in Figure 2 where cost values are of type `text` and `sum` has the signature `sum: text * text * text -> fact`. In the same manner, the comparison between cost values is defined with the fact `less`.

The existing AVANTSSAR model checkers have limited support for numeric values. Using SMT-based techniques would allow for integer costs and a better evaluation of complex attacks such as distributed DoS, where a more sensitive cost analysis must be done. For example, the initial cost of the adversary can be high, but it can be alleviated over multiple protocol sessions. Only a few manual analyses have been done with explicit numeric cost values [15]; most analyses in the literature are symbolic, using a monoid as cost structure.

Transitions can be easily augmented by costs. This has to be done for both protocol steps (as described in detail in [11]) and intruder deductions. Figure 2 shows the cost definition and an intruder deduction modeled as protocol transition that keeps track of cost. The example is a deduction in which the intruder performs a signature with key Y over term X , denoted by $\text{costSig}(X, Y)$, and incurring cost `high`, with the initial condition that he knows both X and Y .

```

sum(low, low, low).
sum(low, medium, medium).
sum(medium, low, medium).
sum(low, high, high).
sum(high, low, high).
sum(high, medium, high).
sum(medium, high, high).
sum(high, high, high).
less(low, medium).
less(medium, high).
less(low, high)

step trans_1(X, Y, Cost, NewCost, ID):=
  state_adv(i, ID, 0)
  .iknows(X).iknows(Y)
  .cost(i, Cost)
  .sum(Cost, high, NewCost)
=>
  state_adv(i, ID, 0)
  .iknows(costSig(X,Y))
  .cost(i, NewCost)
  .sum(Cost, high, NewCost)

```

Fig. 2. Defining costs (left) and a cost-augmented transition for a signature (right)

3.2 Defining the Attack Condition

To flag an attack on principal P , a necessary condition is that the intruder cost is less than the cost incurred by P : $\text{cost}(i, C_i). \text{cost}(i, C_p). \text{less}(C_i, C_p)$. In addition, for abusive use, we need to keep track of the protocol initiator. This can be done by augmenting the initial transition of the protocol (done by principal A) with the fact $\text{initiate}(A)$ and adding $\text{initiate}(i)$ in the attack condition (the attack must be repeatable by the intruder). For malicious use, we track the violation of injective agreement. This can be done by augmenting the right-hand side of each send and receive transition with the facts $\text{send}(S, R, M, L, ID)$ and respectively $\text{recv}(S, R, M, L, ID)$, having as parameters the sender, recipient, content, protocol step and instance. The attack is flagged by checking satisfiability of the condition $\text{recv}(S, R, M, L, ID). \text{not}(\text{send}(S, R, M, L, ID))$, which means that a message receive does not have a matching send.

These attack conditions can be further refined along other criteria, such as determining whether the attack is detectable or not by a given principal, or by any honest principal, etc. A more difficult issue is handling costs over multiple

sessions. In this case, principals must not cumulate costs from correct protocol runs, but only from sessions initiated by the adversary or from malicious sessions. This requires rewriting each protocol transition in several ways, keeping track of these conditions, and tracking costs either per-session or per-principal. Modelling details are given in [11].

As an example, we discuss the Station-to-Station protocol (STS) [8] depicted in Figure 3. The protocol computes a shared session key $k = \alpha^{xy}$ starting from the random values x and y chosen by the two participants.

$$\begin{array}{ll}
 A \rightarrow B : \alpha^x & A \rightarrow \text{Adv}(B) : \alpha^x \\
 B \rightarrow A : \alpha^y, \text{Cert}_B, E_k(\text{sig}_B(\alpha^y, \alpha^x)) & \text{Adv} \rightarrow B : \alpha^x \\
 A \rightarrow B : \text{Cert}_A, E_k(\text{sig}_A(\alpha^x, \alpha^y)) & B \rightarrow \text{Adv} : \alpha^y, \text{Cert}_B, E_k(\text{sig}_B(\alpha^y, \alpha^x)) \\
 & \text{Adv}(B) \rightarrow A : \alpha^y, \text{Cert}_B, E_k(\text{sig}_B(\alpha^y, \alpha^x)) \\
 & A \rightarrow \text{Adv}(B) : \text{Cert}_A, E_k(\text{sig}_A(\alpha^x, \alpha^y))
 \end{array}$$

Fig. 3. Station to Station protocol (left) and Lowe’s attack (right)

Lowe’s attack [13], in the right part of Figure 3, shows the adversary capturing the message sent by A to B and resending it in his own name to B . Afterwards, B is talking to Adv , while A believes she is talking to B . $\text{Adv}(B)$ means the adversary impersonating B , while Adv is the adversary acting as himself.

The attack found by Lowe shows a flaw in the protocol, irrespective of costs. Later, Meadows [16] analyzed this attack from a cost-based perspective. Our model allows a model checker to detect this attack automatically. By using an attack condition (`attack_state`) such as

```

dos_on_a(X, Y, P, V, L, ID) := cost(a, X).cost(i, Y).less(Y, X)
    .recv(P, a, V, L, ID) & not(send(P, a, V, L, ID))

```

we direct the model checker to find a protocol trace in which the adversary has lower cost than the honest principal A , who accepts a message from a different session. Figure 4 presents the attack trace found by CL-Atse (release 2.5-8). The attack differs slightly from the one found by Lowe, but by placing different constraints the back-end can reproduce Lowe’s attack as well. The trace shows the adversary reusing a value sent by A to obtain a response from B that is further redirected and accepted by A . Steps 1 to 3 are from A ’s session with B (compromised by the intruder in step 3), while step 2’ is from a session between the intruder and B (step 1’ is implicit since the intruder knows everything sent over the network). The cost of both A and B is *high* as they compute modular exponentiations while their beliefs about the resulting shared session key (α^{xy}) are both wrong. A believes she shares a key with B , while B believes he shares the key with Adv , who is actually unable to compute it without knowing x .

The cost of the adversary is *low* as he doesn’t perform computations except for redirecting messages, which is assumed to be cheap.

$$\begin{array}{l}
1. A \rightarrow B : \alpha^x \\
2. B \rightarrow I(A) : \alpha^{y_1}, Cert_B, \\
\quad E_k(sig_B(\alpha^{y_1}, \alpha^x)) \\
2'. B \rightarrow I : \alpha^{y_2}, Cert_B, \\
\quad E_k(sig_B(\alpha^{y_2}, \alpha^x)) \\
3. I(B) \rightarrow A : \alpha^{y_2}, Cert_B, \\
\quad E_k(sig_B(\alpha^{y_2}, \alpha^x))
\end{array}
\left\{
\begin{array}{l}
i \rightarrow (A, 5) : \{\} \\
(A, 5) \rightarrow i : costExp(g, n5(XA)) \\
\& \text{Built from trans0} \\
i \rightarrow (B, 12) : costExp(g, n5(XA)) \\
(B, 12) \rightarrow i : costExp(g, n6(XB)).certB. \\
|costExp(g, n6(XB)).costExp(g, n5(XA))_inv(b_pk)| \\
_costExp(costExp(g, n5(XA)), n6(XB)) \\
\& \text{Built from trans1} \\
i \rightarrow (B, 22) : costExp(g, n5(XA)) \\
(B, 22) \rightarrow i : costExp(g, n14(XB)).certB. \\
|costExp(g, n14(XB)).costExp(g, n5(XA))_inv(b_pk)| \\
_costExp(costExp(g, n5(XA)), n14(XB)) \\
\& \text{Built from trans1} \\
i \rightarrow (A, 5) : costExp(g, n14(XB)).certB. \\
|costExp(g, n14(XB)).costExp(g, n5(XA))_inv(b_pk)| \\
_costExp(costExp(g, n5(XA)), n14(XB)) \\
(A, 5) \rightarrow i : certA. \\
|costExp(g, n5(XA)).costExp(g, n14(XB))_inv(a_pk)| \\
_costExp(costExp(g, n14(XB)), n5(XA)) \\
\& \text{Built from trans2}
\end{array}
\right.$$

Fig. 4. STS and the corresponding attack trace shown by CL-Atse

4 Combining Transitions and Horn Clauses for Detection of Guessing Attacks

A guessing attack is done by guessing the value of a low-entropy secret and being able to verify the guess. This can be done in two distinct cases. The first case requires knowing the image of a one-way function on the secret, e.g., $h(s)$, and being able to compute the function on arbitrary values for verification. The second case computes the output of a function inverse that depends on the secret (e.g., the secret is part of a decryption key), and checks a property of the output, e.g., a known part a in decrypting $\{a.m\}_s$ or two equal parts for $\{m.m\}_s$. We have presented a formalization and automation of this approach in [10].

Related work on guessing is based on various theories. A widely used definition is due to Lowe [14], while Abadi et al. [1] present a sound approach from an algebraic point of view based on indistinguishability. Corin et al. [6] also use equational theories, while [12] explicitly represents intruder computation steps, but is limited to offline attacks. Tools that are able to find guessing attacks are presented by Blanchet [5], Corin et al. [7], and Lowe [14].

4.1 Formalization of Guessing

Our approach uses oracles to represent terms obtained as functions computed over the secret. An adversary may establish two kinds of relations with an oracle: *observes* means that the adversary knows the output of the oracle, for some inputs which are not necessarily known. The stronger notion of *controls* means

that the adversary can apply inputs of its choice to the oracle and obtain the resulting output. Thus, *controls* implies *observes*, but the reverse does not hold.

With these notions defined, the first case, when the adversary knows the image of a one-way function over the secret and controls the corresponding oracle, is verified using the guessing rule:

$$\text{observes}(O^f(s)) \wedge \text{controls}(O^f(s)) \Rightarrow \text{guess}(s) \quad (1)$$

Here $O^f(s)$ denotes the oracle computing the function (term) f over secret s . Guessing holds because the adversary knows the image of a one-way function over the secret and since he controls the oracle for that function he can successively give all values of the secret as input to the oracle then verify the result. This rule can detect guessing attacks on terms such as $m.h(m, s)$ or $\{s\}_s$, etc.

In the second case, when the secret is part of the key of an invertible function we need to check if a term is verifiable by the adversary. We define a term as *verifiable* by the adversary in any of the following cases:

- the term is already known:

$$\text{verifiable}(Term) :- \text{iknows}(Term) \quad (2)$$

- the term is a digital signature, and the public key and message are known:

$$\text{verifiable}(\text{apply}(\text{inv}(PK), Term)) :- \text{iknows}(PK) \wedge \text{iknows}(Term) \quad (3)$$

- the image of a one-way function on the term and a controllable oracle for that function are known:

$$\begin{aligned} \text{verifiable}(Term) :- \text{iknows}(\text{apply}(F, T')) \wedge \text{part}(Term, T') \\ \wedge \text{controls}(O^{F(T')}(Term)) \end{aligned} \quad (4)$$

- the term is an encryption with a controllable decryption oracle, and some part T'' of the encrypted term is verifiable if the remaining part T' is added to the intruder knowledge (expressed by fact $\text{split}(Term, T', T'')$):

$$\begin{aligned} \text{verifiable}(\text{script}(K, Term)) :- \text{controls}(O^{\{M\}_{K^{-1}}}(M)) \\ \wedge \text{split}(Term, T', T'') \wedge \text{verifiable}(T'') \end{aligned} \quad (5)$$

We can now define the second case of guessing. If the adversary *observes* an encryption oracle that uses a key dependent on the secret s , if he *controls* the corresponding decryption oracle (with s as input) and can *verify* a part of the encrypted message then the adversary can guess the secret, i.e.,

$$\begin{aligned} \text{observes}(O^{\{Term\}_K}(s)) \wedge \text{controls}(O^{\{Term\}_{K^{-1}}}(s)) \\ \wedge \text{split}(Term, T', T'') \wedge \text{verifiable}(T'') \Rightarrow \text{guess}(s) \end{aligned} \quad (6)$$

For example, consider h , m_1 , $\text{apply}(h, \text{pair}(m_1, m_2))$ and $\text{script}(s, m_2)$ as known. Since m_2 is a part of $\text{apply}(h, \text{pair}(m_1, m_2))$ and this function is controllable

in m_2 (h and m_1 being known), then m_2 is verifiable by rule [4](#). Knowing $\text{scrypt}(s, m_2)$ and controlling the decryption oracle, one can guess s .

Next, consider that $\text{scrypt}(s, \text{scrypt}(k, \text{pair}(m, m)))$ and k are known. Using rule [5](#), $\text{scrypt}(k, \text{pair}(m, m))$ is verifiable, since adding the first instance of m to the intruder knowledge, one can verify the second m from the pair. This in turn allows guessing s . On the other hand, from $\text{scrypt}(s, \text{scrypt}(k, m))$ and k the intruder cannot in general guess s : if symmetric encryption is done with a one-time pad then the decrypted m has no meaning and cannot be verified. Of course, if the adversary knows both m and k , the term is verifiable by rule [11](#) since $\text{iknows}(\text{scrypt}(k, m))$ holds. Other verification rules can be added to the model, for example in the case of authenticated encryption, etc.

To implement *observes*, the analysis performed by the model checker needs to determine if a composed term contains the secret to be guessed. We express observing an oracle as $\text{ihears}(T) \wedge \text{contains}(T, S)$, where S is the guessable secret, and *ihears* is a fact added to the right-hand side of each send transition in the protocol, supplementary to *iknows*. This new fact is needed to distinguish terms overheard on the network from terms otherwise derived by the intruder.

To decide *controls*, the analysis takes a term containing the secret and constructs a new term where the secret is replaced by a fresh value. If the adversary knows this term, it *controls* the oracle corresponding to the original term. Controlling an oracle is then modeled by the facts $\text{replace}(T, T\text{New}) \wedge \text{iknows}(T\text{New})$.

For efficiency, the above expressions can be used directly in the attack condition rather than introducing explicit *observes* and *controls* facts. Moreover, a sufficient condition for the case of *offline* guessing is whether the adversary knows all parts of a term except the secret which it tries to guess. This is easier to model and verify by the back-ends and will be used in the following subsections.

We next discuss how to express the facts *contains* and *replace* for checking the presence of a secret in a term and constructing a new term, respectively, and how to efficiently model the verifiability conditions given above using Horn clauses. We evaluate the various approaches on a protocol case study.

4.2 Processing Terms That Contain the Secret

Checking whether a term contains a particular secret, and replacing it with a new value is simple conceptually but challenging in practice. These steps are the basis for detecting simple attacks such as those covered by the first guessing case of rule [11](#), e.g., guessing s if a and $h(a.s)$ are known. We discuss three ways of implementing containment and replacement, with major differences both in writing the rules and in the analysis time to find an attack.

We test the various approaches using as example the MS-CHAP protocol [\[20\]](#), depicted in [Figure 1](#). To make analysis times more significant we modify this simple protocol by inserting more complex terms. In MS-CHAP* the last term, $H(k_{AB}, N_A)$, is changed to $H(N_A, k_{AB}, N_A)$, while for MS-CHAP** we concatenate N_A to the key seven times: $H(N_A, \dots, N_A, k_{AB}, N_A)$. While guessing is conceptually straightforward in all cases, for complex terms the rules require more processing time, highlighting the differences between the approaches.

Naive Approach with Transitions

The straightforward approach is to consider terms heard by the intruder on the network, to define *contains* and *replace* for atoms, and to derive them for composed terms using the corresponding facts for their components. For atomic terms, rules can be defined as shown in the left part of [Figure 5](#) for an atom of type *text*. Since the atom is not equal to the secret, *replace* does not change it, and *contains* is asserted with a dummy null secret.

For composed terms, *contains* and *replace* facts can be derived if these facts are known for their components. This is shown in the right part of [Figure 5](#) for a term composed with *pair*, and similar rules are defined for terms composed with *script*, *crypt*, etc. In our implementation, we restrict the application of these rules only for terms sent over the network, in order to avoid their inefficient application over the entire intruder knowledge. This is because the set of terms known by the intruder is large, conceptually even unbounded (since the intruder can always create fresh terms or compose already known terms with known operators), and the back-end may fail to terminate. Using *ihears* instead of *iknows* also allows us to derive the sub-terms of terms that are heard on the network, which can be easily done by adding transitions for each type of composed term.

In practice, this modelling approach works for decomposing simple terms, however if the terms are more complex, and many transitions are needed, then the back-end times out attempting to verify the model. As seen in [Table 1](#), this modelling variant fails for terms on which the forthcoming procedures succeed.

<pre>state_process(A, ID, 0) .ihears(AtomText) .not(equal(AtomText, s)) => state_process(A, ID, 0) .ihears(AtomText). .contains(AtomText, snull) .replace(AtomText, AtomText)</pre>	<pre>state_process(A, ID, 0) .ihears(pair(T1, T2)) .contains(T1, S1).contains(T2, S2) .replace(T1, T1New).replace(T2, T2New) => state_process(A, ID, 0) .ihears(pair(T1, T2)) .contains(T1, S1).contains(T2, S2) .replace(T1, T1New).replace(T2, T2New) .contains(pair(T1, T2), S1) .contains(pair(T1, T2), S2) .replace(pair(T1, T2), pair(T1New, T2New))</pre>
--	---

Fig. 5. Contains and replace for atomic terms (left) and composed terms (right)

Improved Approach with Transitions

The previous approach is inefficient because the steps for customized intruder deductions can be interleaved with protocol steps, leading to exponential complexity. To avoid this, we control the order in which the terms are processed by placing them in a stack (constructed with *pair* and a dummy separator). Terms are processed by structural decomposition and each new sub-term is placed on top of the stack unless it is an atom and the *contains* and *replace* facts for it can

be directly deduced. Clearly an atom contains the secret if and only if it is the actual secret, otherwise *contains* is false and *replace* leaves the atom unchanged.

For example, consider the term $script(k, h(pair(N_A, N_B)))$ heard over the network. In the first step the stack contains only this term. Next, k (the left operand of *script*) is added to the top of the stack. As this operand is atomic, one can directly establish *contains* and *replace* for it, and remove it from the stack. The next item on the stack will be the right operand of *script*, i.e., $h(pair(N_A, N_B))$. The next element is $pair(N_A, N_B)$, with the stack now containing three items, and so on. On the left side of [Figure 6](#) an atom of type *text* is eliminated from the list, while on the right side a composed term is split into its components. This mechanism greatly reduces complexity due to interleaving.

As can be seen in [Table 1](#), this modelling variant succeeds in deriving the guess also for the artificially complicated structure of MS-CHAP**. However, its time requirement increases significantly with the complexity of the term, a drawback removed in the next modelling solution.

<pre>state_process(A, ID, 1) .process(pair(pair(AtomText, sep), Right)) .not(guessable(AtomText)) => state_process(A, ID, 0) .process(Right) .contains(AtomText, snull). .checked(AtomText) .replace(AtomText, AtomText). .replaced(AtomText)</pre>	<pre>state_process(A, ID, 1) .process(pair(pair(pair(T1, T2), sep), Right)) .contains(T1, S1).replace(T1, T1New) .contains(T2, S2).replace(T2, T2New) => state_process(A, ID, 0) .process(Right) .contains(T1, S1).replace(T1, T1New) .contains(T2, S2).replace(T2, T2New) .checked(pair(T1, T2)) .contains(pair(T1, T2), pair(S1, S2)) .replace(pair(T1, T2), pair(T1New, T2New)) .replaced(pair(T1, T2)).</pre>
---	---

Fig. 6. Improved contains/replace for atomic terms (left) and composed terms (right)

Efficient Approach with Horn Clauses

Horn clauses are more elegant and intuitive for modelling intruder deductions. They were specially introduced in ASLan for this purpose, as well as for modelling static or dynamic security policies.

The model checkers of the AVANTSSAR platform process Horn clauses in different ways, depending on their overall exploration strategy. CL-Atse employs a backward search, using Horn clauses only when deriving some fact is required (e.g., for the left-hand side of a transition). SATMC on the other hand employs a forward strategy and saturates the set of known facts by transitively applying Horn clauses after each transitions. Thus, Horn clauses written with one search strategy in mind may lead a model checker employing the opposite strategy to non-termination. We have devised models adapted to the use of CL-Atse.

For example, the Horn clauses in [Figure 7](#) find both the part of a term and its remainder. The fact $ispart(T_1, T_2, T_3)$ denotes that T_1 is split into disjoint parts T_2 and T_3 . The Horn clause $part_left$ states that T_2 is part of $pair(T_0, T_1)$ with remainder $pair(T_0, T_3)$ if T_2 is part of T_1 with remainder T_3 . Such rules need to be written for all operators that can be applied on terms.

```

hc part_null(T1) :=
    ispart(T1, null, T1)
hc part_id(T1) :=
    ispart(T1, T1, null)
hc part_left(T0, T1, T2, T3) :=
    ispart(pair(T0, T1), T2, pair(T0, T3)) :- ispart(T1, T2, T3)
hc part_right(T0, T1, T2, T3) :=
    ispart(pair(T0, T1), T2, pair(T3, T1)) :- ispart(T0, T2, T3)
hc part_scrypt_left(T0, T1, T2, T3) :=
    ispart(scrypt(T0, T1), T2, pair(T0, T3)) :- ispart(T1, T2, T3)
hc part_scrypt_right(T0, T1, T2, T3) :=
    ispart(scrypt(T0, T1), T2, pair(T3, T1)) :- ispart(T0, T2, T3)

```

Fig. 7. Splitting terms using Horn clauses

1. $A \rightarrow B: A$	{	$i \rightarrow (\text{chap_Init}, 11) : \text{start}$
		$(\text{chap_Init}, 11) \rightarrow i : a$
2. $B \rightarrow A: N_B$		$i \rightarrow (\text{chap_Resp}, 18) : a$
		$(\text{chap_Resp}, 18) \rightarrow i : n4(Nb)$
3. $A \rightarrow B: N_A,$ $H(k_{AB}, N_A, N_B, A)$	{	$i \rightarrow (\text{chap_Init}, 13) : n4(Nb)$
		$(\text{chap_Init}, 13) \rightarrow i : \text{pair}(n2(Na), h(\text{pair}(s,$ $\text{pair}(n2(Na), \text{pair}(Nb(2), a))))))$
4. $B \rightarrow A: H(k_{AB}, N_A)$	{	$i \rightarrow (\text{chap_Resp}, 20) : \text{pair}(n2(Na), h(\text{pair}(s,$ $\text{pair}(n2(Na), \text{pair}(n4(Nb), a))))))$
		$(\text{chap_Resp}, 20) \rightarrow i : h(\text{pair}(s, n2(Na)))$
		$\text{controls}(h(\text{pair}(s, n2(Na))), s),$ $\text{iguess}(s),$ $\text{ihears}(h(\text{pair}(s, n2(Na))))),$
Horn clause facts:		$\text{ispart}(h(\text{pair}(s, n2(Na))), h(\text{pair}(s, n2(Na))), \text{null}),$ $\text{ispart}(s, \text{pair}(s, n2(Na)), \text{pair}(n2(Na), \text{null})),$ $\text{ispart}(s, s, \text{null}),$ $\text{observes}(h(\text{pair}(s, n2(Na))), s)$

Fig. 8. MS-CHAP v2 and the corresponding attack trace found by CL-Atse

In the attack trace from [Figure 8](#) the intruder was forced to guess k_{AB} from the message in step 4, although it could have also guessed at step 3. The Horn clauses show that the intruder *observes* the term from step 4, and repeatedly applies rules involving *ispart* until it can derive *controls*, which then allows the guess. Rules for *observes* and *controls* are discussed in the next subsection.

[Table 1](#) shows that by using this approach the increase in time requirements is negligible for more complex terms which otherwise require several seconds of processing, or even fail if naive transition rules are used for term processing.

Table 1. Timing results for attack detection on MS-CHAP with CL-Atse

	MS-CHAP	MS-CHAP*	MS-CHAP**
Naive Transitions	456 ms	820 ms	TOUT
Efficient Transitions	1272 ms	1812 ms	10529 ms
Horn Clauses	120 ms	120 ms	112 ms

4.3 Using Horn Clauses and Transitions for Intruder Deductions

Finally, to flag a guessing attack, we need to determine whether some term is verifiable by the intruder. [Figure 9](#) shows rules for the verifiability conditions discussed previously. To achieve this, in some cases we need to add terms to the intruder knowledge as shown in [Figure 10](#). This is important for modelling: while Horn clauses can be used for verifying terms, they cannot be used to add terms to intruder knowledge when working with CL-Atse, due to the backward strategy it employs when using Horn clauses. (SATMC however can do this, as it due to the forward strategy employed). The two guessing cases are detected by the Horn clauses in [Figure 11](#). Thus, to validate the guess we have to use a mixture of Horn clauses and intruder transitions. Using these, CL-Atse is able to detect guessing from terms such as $\{m, m\}_s$ or $k, \{\{m, m\}_k\}_s$, etc.

```

% verify known term
hc verif_iknows(MsgA) :=
    verifiable(MsgA) :- iknows(MsgA)

% verify signature
hc verif_sign(PbK, MsgA) :=
    verifiable(apply(inv(PbK), MsgA)) :- iknows(PbK), iknows(MsgA)

% verification of term under hash
hc verif_hash(MsgA, MsgB, MsgC) :=
    verifiable(MsgA) :- iknows(apply(h,MsgB)),
                        ispart(apply(h,MsgB), MsgA, MsgC), iknows(MsgC)

% the ciphertext is verifiable if the encryption key is known
% and part of the plaintext is verifiable
hc verif_scrypt_ciphertext(K, MsgA, MsgB, MsgC) :=
    verifiable(scrypt(K, MsgA)) :- iknows(K), split(MsgA, MsgB, MsgC),
    verifiable(MsgC)

```

Fig. 9. Horn clauses for verifying terms

```

% split a message if it was not split before
step trans_split(A, MsgA, MsgB, MsgC, K) :=
  state_split(A)
  .ihears(scrypt(K, MsgB)).ispart(MsgB, MsgA, MsgC)
  .not(equal(MsgC, null)).not(is_split(MsgB))
=>
state_split(A)
.ihears(scrypt(K, MsgB)).ispart(MsgB, MsgA, MsgC)
.iknows(MsgA)
.split(MsgB, MsgA, MsgC).is_split(MsgB)

```

Fig. 10. Transition for adding terms to intruder knowledge

```

hc controls_hash(S, K, Rest, Msg) :=
controls(apply(h, Msg), S) :- ihears(apply(h, Msg)),
                             ispart(S, Msg, Rest), iknows(Rest)

hc observes_hash(S, K, Rest, Msg) :=
observes(apply(h, Msg), S) :- ihears(apply(h, Msg)),
                              ispart(S, Msg, Rest)

hc guess_case_i(S, Msg) :=
iguess(S) :- lowentropy(S), observes(Msg, S), controls(Msg, S)

hc controls_scrypt(S, K, KRest, Msg) :=
controls(scrypt(K, Msg), S) :- ihears(scrypt(K, Msg)),
                              ispart(S, K, KRest), iknows(KRest)

hc observes_scrypt(S, K, KRest, Msg) :=
observes(scrypt(K, Msg), S) :- ihears(scrypt(K, Msg)),
                              ispart(S, K, KRest)

hc guess_case_ii(S, K, MsgA, MsgB, MsgC) :=
iguess(S) :- lowentropy(S),
             observes(scrypt(K, MsgA), S), controls(scrypt(K, MsgA), S),
             split(MsgA, MsgB, MsgC), verifiable(MsgC)

```

Fig. 11. Horn clauses for guessing

4.4 Distinguishing Detectable from Undetectable On-line Attacks

With the guessing mechanism established above, the attack condition can be stated in different flavours. For example, as the deduction rules allow detecting on-line attacks, we can ask whether the attack is detectable or not by some (or any) honest participant. The relevance of this kind of undetectable on-line attacks was previously pointed out by Ding and Horster [9].

We can express that guessing is undetectable for honest participants if for all executions where guessing happens, the protocol is completed normally by all participants. Thus, we can reformulate undetectable guessing as a reachability

check for an attack state in which the secret has been guessed *and* all protocol participants have completed execution.

In ASLan models, each participant has a unique identifier ID which is part of its **state** fact. We also define for each participant the fact **running**(ID) which is true in every state except the participant's initial and final states. An adversary *observes* (*controls*) an oracle *undetectably* if it *observes* (*controls*) the oracle and all protocol participants reach a final state, i.e., no fact **running**(ID) holds.

A protocol description can be automatically augmented to allow for this check by statically identifying its *initial* and *final* transitions. Initial transitions have in the LHS a *state* fact, whereas *final transitions* have in the RHS a *state* fact that does not appear in the LHS of any other transition rule. Every initial transition is augmented to generate a fresh ID value, and the positive fact **running**(ID) is added to its RHS. Every *final transition* is augmented with the fact **running**(ID) on the LHS, but not on the RHS, thus it becomes false. This protocol adaptation allows to directly express undetectable guessing.

The same technique can be used to distinguish offline attacks. This is achieved by checking for attacks, while requiring that no fresh ID is ever generated. It may also be useful to check, for instance, if only adversary *observe* actions were done on-line, while *controls*, which involves computations and is more tedious, is performed offline. This can be done by checking that no fresh ID is generated between *observes* and *controls*. Thus, our approach allows not only the detection of guessing attacks, but also their classification.

5 Conclusions

As model checkers for security protocols do not by default support the detection of all attacks, one needs to use customized intruder deductions and transitions for this purpose. This allows the handling of new types of attacks without changing the model-checking back-ends. In this paper, we have explored two such case studies: modelling guessing attacks and denial of service by resource exhaustion. These attacks are relevant as many protocols used in practice are vulnerable to them, and we show the applicability of our theories with automatically obtained attack traces on known protocols.

We present different modelling options and investigate the relative efficiency of transition rules and Horn clauses, with the latter providing significant performance gain and allowing the processing of more complex message terms. The modelling approaches described here show the power of the ASLan specification language which serves as input to the AVANTSSAR model checkers.

We hope that the approaches shown here can provide a starting point for modelling other types of attacks that are currently not detected by default.

References

1. Abadi, M., Baudet, M., Warinschi, B.: Guessing attacks and the computational soundness of static equivalence. In: Aceto, L., Ingólfssdóttir, A. (eds.) FOSSACS 2006. LNCS, vol. 3921, pp. 398–412. Springer, Heidelberg (2006)

2. Armando, A., Compagna, L.: SAT-based model-checking for security protocols analysis. *International Journal of Information Security* 7(1), 3–32 (2008)
3. AVANTSSAR: Deliverable 2.3 (update): ASLan++ specification and tutorial (2011), <http://www.avantssar.eu>
4. Basin, D.A., Mödersheim, S., Viganò, L.: OFMC: A symbolic model checker for security protocols. *Internat. J. of Information Security* 4(3), 181–208 (2005)
5. Blanchet, B.: An efficient cryptographic protocol verifier based on Prolog rules. In: *Proc. 14th IEEE Computer Security Foundations Workshop*, pp. 82–96 (2001)
6. Corin, R., Doumen, J.M., Etalle, S.: Analysing password protocol security against off-line dictionary attacks. In: *Proc. 2nd Int'l. Workshop on Security Issues with Petri Nets and other Computational Models (WISP)*, pp. 47–63 (2004)
7. Corin, R., Malladi, S., Alves-Foss, J., Etalle, S.: Guess what? Here is a new tool that finds some new guessing attacks. In: *Proc. Workshop on Issues in the Theory of Security*, pp. 62–71 (2003)
8. Diffie, W., van Oorschot, P.C., Wiener, M.J.: Authentication and authenticated key exchanges. *Designs, Codes and Cryptography* 2(2), 107–125 (1992)
9. Ding, Y., Horster, P.: Undetectable on-line password guessing attacks. *Operating Systems Review* 29(4), 77–86 (1995)
10. Groza, B., Minea, M.: A formal approach for automated reasoning about off-line and undetectable on-line guessing (short paper). In: Sion, R. (ed.) *FC 2010. LNCS*, vol. 6052, pp. 391–399. Springer, Heidelberg (2010)
11. Groza, B., Minea, M.: Formal modelling and automatic detection of resource exhaustion attacks. In: *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, ASIACCS (2011)*
12. Hanks Drielsma, P., Mödersheim, S., Viganò, L.: A formalization of off-line guessing for security protocol analysis. In: Baader, F., Voronkov, A. (eds.) *LPAR 2004. LNCS (LNAI)*, vol. 3452, pp. 363–379. Springer, Heidelberg (2005)
13. Lowe, G.: Some new attacks upon security protocols. In: *Proc. of the 9th IEEE Computer Security Foundations Workshop*, pp. 162–169 (1996)
14. Lowe, G.: Analysing protocols subject to guessing attacks. *Journal of Computer Security* 12(1), 83–98 (2004)
15. Matsuura, K., Imai, H.: Modification of internet key exchange resistant against denial-of-service. In: *Pre-Proceedings of Internet Workshop*, pp. 167–174 (2000)
16. Meadows, C.: A cost-based framework for analysis of denial of service networks. *Journal of Computer Security* 9(1/2), 143–164 (2001)
17. Ramachandran, V.: Analyzing DoS-resistance of protocols using a cost-based framework. *Tech. Rep. DCS/TR-1239*, Yale University (2002)
18. Smith, J., González Nieto, J.M., Boyd, C.: Modelling denial of service attacks on JFK with Meadows's cost-based framework. In: *Proc. of the 4th Australasian Information Security Workshop*, pp. 125–134 (2006)
19. Turuani, M.: The CL-Atse protocol analyser. In: Pfenning, F. (ed.) *RTA 2006. LNCS*, vol. 4098, pp. 277–286. Springer, Heidelberg (2006)
20. Zorn, G.: Microsoft PPP CHAP extensions, version 2 (2000)

Improving the Usability of Specification Languages and Methods for Annotation-Based Verification^{*}

Bernhard Beckert, Thorsten Borner, and Vladimir Klebanov

Institute for Theoretical Computer Science,
Karlsruhe Institute of Technology, Germany

<http://formal.itl.kit.edu>

Abstract. It is widely recognized that human input is indispensable in deductive verification of real-world code. Verification engineers have to guide the proof search and provide information reflecting their insight into the workings of the program. Lately we have seen a shift towards an annotation-based paradigm – sometimes called “verifying compiler” –, where this information is provided in the form of program annotations instead of interactively during proof construction.

Suspensions have been growing recently that expressing verification knowledge as annotations in their current form suffers from serious scalability and maintainability issues.

In this paper, we pinpoint some of the biggest neuralgic spots and provide recommendations to the designers of annotation-based verification systems aimed to improve usability of specification languages and methods and, thus, the tool’s productivity. We clarify the different purposes that annotations can serve and show why a certain class of annotations that are not program requirements is currently indispensable for proof construction. Moreover, we discuss how the use of data abstractions can be improved in annotation-based specifications.

1 Introduction

Program annotations as a form of interaction with the software verification tool have several important advantages. They make verification attempts self-contained, as human guidance is captured textually (and typically in terms closely related to the program at hand). They also keep the program and the specification close to each other, which is helpful as both unavoidably (co-)evolve.

At the same time, suspicions have been growing recently that expressing verification knowledge as annotations in their current form suffers from serious scalability and maintainability issues. This is part of a more general concern that writing specifications may turn into a bottle-neck for program verification (this observation was also, e.g., made in [13]).

^{*} Work partially funded by the German Federal Ministry of Education and Research (BMBF) in the framework of the Verisoft XT project under grant 01 IS 07008 H. The responsibility for this article lies with the authors.

The problems with annotation-based verification arise as the lines between (a) requirement specification, (b) auxiliary information needed for proof construction, and (c) information for proof-search guidance get blurred. Even worse, related specification parts get dispersed and different levels of abstraction inter-mixed.

In this paper, we pinpoint some of the biggest neuralgic spots and provide recommendations to the designers of annotation-based verification systems aimed to improve usability of specification languages and methods and, thus, the tool’s productivity.

After an introduction to the typical architecture and working cycle of annotation-based verification systems (Sect. 2), we discuss in Section 3 the different purposes that annotations can serve, based on a clarification of what the notion of completeness means in this framework. We show why a certain class of annotations that are not program requirements is currently indispensable for proof construction. Users are often surprised that they cannot omit certain non-requirement annotations even for the simplest (sub-)problems. We plead that they must be better educated about the inner workings of a verification system and what kinds of annotations are indispensable in which situations. This is in conflict with the desire to enable the user to work with the verification system as a “black box”, which is generally seen as an important feature of the verifying compiler paradigm.

In Section 4, we discuss how the use of data abstractions can be improved in annotation-based specifications. The lack of syntactical separation between the (abstract) requirements and the non-requirement annotations obscures the inner structure of specifications and makes them hard to understand and maintain.

Finally, in Section 5 we draw conclusions from our research and discuss future work.

The problems we discuss are not inherent to annotation-based verification (nor the verifying compiler approach), but rather due to the currently implemented design decisions of such verification systems. All of the issues mentioned in this work can be overcome by extending specification languages and methodologies. The general, central idea of annotating programs at source-code level – using a language whose syntax is closely related to the programming language – is not the source of the problems described in this paper.

2 Inside a Typical Annotation-Based Verification System

Tools following the annotation-based paradigm include Spec# [1], VCC [12], Caduceus [7] and others. They are all based on powerful fully-automatic provers and decision procedures, and they support real-world programming languages such as C and C#.

Compared to fully automatic verification approaches like model checking or abstract interpretation that need no human interaction, the annotation-based paradigm allows for full functional verification of programs. But the former methods are either restricted in the expressiveness of specifications, the precision of results, or in the kind of programs that can be verified.

In the following we describe the process of software verification with the Verifying C Compiler (VCC). Our observations (unless noted otherwise) are, however, not restricted to this particular setup.

2.1 Structure of the Toolchain

The VCC toolchain allows for modular verification of C programs using method contracts and invariants over data structures. Method contracts are specified by pre- and postconditions. These contracts and invariants are stored as annotations within the source code in a way that is transparent to the regular, non-verifying compiler.

As most annotation-based verification systems today, VCC works using an internal two-stage process. The reason for this is a better separation of concerns and easy integration of different tools. We will discuss the interplay of the two stages, but many of our remarks also apply to one-stage or multi-stage approaches.

The first stage of the VCC toolchain translates the annotated C code into first-order logic via an intermediate language called BoogiePL [6]. BoogiePL is a simple imperative language with embedded assertions. From this BoogiePL representation, it is easy to generate a set of first-order logic formulas, which state that the program satisfies the assertions. These formulas are called verification conditions and the stage a verification condition generator (VCG).

In the second stage, the resulting formulas are given to an automatic theorem prover (TP) resp. SMT solver (in our case Z3 [5]) together with a background theory capturing the semantics of C's built-in operators, etc. The prover checks whether the verification conditions are entailed by the background theory. Entailment implies that the original program is correct w.r.t. its specification.

See Sections 3 and 4 for some real-world examples for specification and verification of C programs with VCC.

2.2 The Possible Outcomes of Invoking an Annotation-Based Verification Tool

In practice, where the limitations of resources are relevant, the possible outcomes of a verification attempt using a two-stage annotation-based verification system are:¹

1. The formulas generated by the VCG are valid, and the TP has found a proof for that. This outcome entails that the original program has the specified properties.
2. Some generated formula is not valid, and the TP has found a counterexample. This can mean two things: (a) The program is not correct w.r.t. its specification, i.e., there is an error in either the program code or the specification. (b) The program satisfies the specification, but some loop invariant

¹ We assume that the programs to be verified are of reasonable size such that only the theorem proving stage can run out of resources and not the VCG stage.

or other auxiliary annotation is missing or not strong enough and, as a consequence, some generated verification condition is not a valid formula. We will discuss this distinction in more detail in Section 3.1.

3. The TP runs out of resources (time or space). This can mean two things:
 - (a) The generated formula is valid and the program is correct (as in Case 1 above), but the TP could not find a proof in the allotted time/space. (b) The formula is not valid (as in Case 2 above), but the TP could not find a counterexample. The non-validity can, again, be due either to the program being incorrect or to some auxiliary annotation being not strong enough.

In Case 1 above, the invocation of the verification system was successful – a desired but rare case in practice. Cases 2 and 3 are much more common, and the user has to analyze the problem. If they find (using the potential counterexample) that the program indeed does not satisfy the specification, the error has to be corrected. If they find that the program satisfies the specification, then new auxiliary annotations (stronger invariants, helpful lemmas, etc.) have to be added. This process is repeated until the program can be verified.

3 Distinguishing Different Kinds of Annotations

3.1 Annotations and Their Properties

Preliminaries. In the following we assume as given a programming language, and an annotation language for expressing specifications. Which annotations are possible depends on the particular language; typical annotations are for example invariants, pre-/postcondition pairs, and assertions of various kinds. In order to easily relate alternative potential annotations for the same program, we take the view that annotations are disjoint from regular program statements. On the other hand, each annotation has an *intended context* (statement, method, class, etc.). We assume that we only deal with combinations of programs and annotations without context mismatches, i.e., annotations are compatible with the programs to which they are added. The context an annotation refers to must actually exist in the program, and the symbols used in the annotation must be defined for that context.

Definition 1 (Combination of program and annotations). *If P is a program and A is a set of annotations compatible with P , then we call the pair $P \vdash A$ the combination of the two. The parameter $<$ fixes the order of annotations if several of them have the same intended context. We will omit the ordering whenever it is irrelevant or clear from the context and simply write $P+A$.*

Definition 2 (Annotation satisfaction). *We assume that there is a definition of when a program P satisfies a specification REQ , denoted by $\models P+REQ$.*

Definition 3 (Strength of annotations). *An annotation A is (logically) stronger than an annotation B , in symbols $A \Rightarrow B$, if $\models P+B$ holds for all programs P with $\models P+A$.*

Different Purposes of Program Annotations. Annotations can serve distinctively different purposes, though sometimes several different ones simultaneously. The following classification of annotations is neither syntactic nor semantic, but concerns rather the pragmatics of their use and the intentions of their author.

Requirement Annotations. Requirement annotations constitute the specification of the program. They assure the behavior of the program (module) towards its environment. They are the reason for performing verification. Typical requirement annotations are pre- and postconditions, class invariants, or resource consumption limits. They are visible externally and cannot be changed easily.

Auxiliary Annotations. Auxiliary annotations are used to guide the proof search. They are usually not part of program requirements. As long as they satisfy their purpose, auxiliary annotations can be changed anytime without notice. We further distinguish two subclasses of auxiliary annotations:

- (a) The first subclass is necessary merely for efficiency reasons. It encompasses lemmas, intermediate assertions, quantifier instantiation triggers, and the like. These annotations are not necessary for completeness. They can always be made obsolete by increasing the space/time available for proof search or by advances in SMT prover technology. Another purpose of annotations from this subclass is to inspect the proof state. For this, the user temporarily adds auxiliary annotations to get information about implicit “knowledge” of the proof system at particular points in the proof search – in order to eventually come up with the right auxiliary annotations needed to complete the proof (as defined in Def. 7).
- (b) The other subclass of auxiliary annotations are essential annotations. Getting them right is essential for completeness, the very existence of a correctness proof. The most prominent essential annotations are loop invariants. Further auxiliary annotations that can be essential are data-structure invariants and abstractions, ownership annotations, and framing conditions.

Monotonicity of Auxiliary Annotations. A very desirable property of annotation satisfaction is monotonicity. Adding auxiliary annotations should strictly increase the strength of the specification, i.e., \models should be monotonic w.r.t. adding annotations.

Definition 4 (Monotonicity of \models). \models is monotonic w.r.t. adding annotations iff, for all programs P and all specifications REQ and AUX the following holds:

$$\text{if } \models P+(REQ \cup AUX) \text{ then } \models P+REQ .$$

In reality, monotonicity of \models w.r.t. adding auxiliary annotations is not given unless we make some restrictions. One concerns assume annotations that add an unchecked assumption to the following proof (and thus can make a specification weaker). Since all proved properties only hold modulo these assumptions,

assume annotations are a correctness risk. For these reasons we always classify assume annotations as part of the requirement and never as auxiliary.

In the same vein, adding a formula to a precondition, and thus weakening it, violates the condition of Def. 4 and is not an acceptable way of adding auxiliary annotations.

3.2 Annotations and Existence of Proofs

To separate the annotations that are essential for the existence of a proof and the ones that are needed only for supporting proof search, one needs a clear understanding of the notion of completeness. In this section, we discuss what completeness means in the framework of annotation-based verification systems and give formal definitions.

Completeness and Relative Completeness. The classical notion of completeness for deduction systems can be adapted to annotation-based verification systems as follows:

Definition 5 (Completeness). *Let S be a verification calculus or system. S is complete if, for any program P satisfying its requirement specification REQ , this fact can be proved using the calculus from a fixed set of axioms Th_S . In symbols:*

$$\text{if } \models P+REQ \text{ then } Th_S \vdash_S P+REQ$$

The semantics of the programming language (used for P) and the annotation language (used for REQ) are encoded in the calculus rules \vdash_S and in the background theory Th_S . The restriction of resources (time and space) of real-world systems is usually not considered for the notion of completeness.

Note also the difference between \models and \vdash : Fewer annotations are easier to satisfy by the program (\models), while more annotations may make it easier to find a proof (\vdash).

Since first-order arithmetics is undecidable, all non-trivial properties of programs are undecidable (Rice's Theorem), and all program verification systems are necessarily incomplete in the sense of Def. 5. Instead the notion of *relative completeness* is used, i.e., completeness in the sense that the system or calculus would be complete if it had an oracle for the validity of formulas about arithmetic [4]. This can be formalized as follows:

Definition 6 (Relative completeness). *A verification system S , consisting of \vdash_S and Th_S , is relatively complete (w.r.t. arithmetics) if, for each program P and specification REQ with*

$$\models P+REQ ,$$

there is a set $Arith$ of valid arithmetical formulas such that

$$Th_S \cup Arith \vdash_S P+REQ .$$

Luckily, undecidability of first-order arithmetics is usually not an obstacle for verification in practice. Experience shows that the axiomatization Th together with the calculus rules of the theorem prover approximate arithmetics well enough and that the valid arithmetic formulas occurring in practice can be derived (which does not imply that finding a derivation is easy or possible automatically but only that a derivation exists). One has to keep in mind, that the distinction between completeness and relative completeness exists, even if the restriction to relative completeness is not a real limitation in practice.

Theoretical Completeness Arguments. Relatively complete calculi exist for many program logics. Harel gives one for first-order Dynamic Logic in [8]. Less-known is the fact that the presence of auxiliary annotations, such as loop invariants, is not a prerequisite for relative completeness.

Harel conducts his relative completeness proof by showing that program logics are no more expressive than first-order arithmetics. That is, for every program there is a first-order arithmetics formula that encodes the same relation between states that the program encodes. The less-known fact is that it is possible to effectively compute such a formula without further input. In fact, Harel’s proof contains a simple algorithm that for any Dynamic Logic formula ϕ effectively computes an equivalent purely first-order arithmetics formula ϕ_A [8, Theorem 3.2]. This construction gives along the way a means to automatically compute the strongest invariant of any loop.

The algorithm is based on Gödelization, i.e., encoding a finite sequence of domain elements into one element. The generated invariant formula asserts the existence of a number encoding a sequence of states corresponding to the forthcoming computation sequence of the loop until it terminates.

Thus, theoretically speaking, the strongest loop invariant for any given loop and so the verification conditions for any given piece of code can be easily computed in polynomial time. That is not a contradiction to general undecidability of program verification, since one undecidable problem (program verification) gets transformed into another undecidable problem (deciding first-order logic with arithmetics). Still, assuming a theoretical standpoint, one can conclude that no auxiliary annotations are really needed because all information contained in annotations can easily be computed by the VCG.

In Practice: Annotation Completeness. The theoretical fact that all necessary annotations can “easily” be constructed (see above) is in practice a red herring because the constructed annotations use Gödelization and, thus, complex arithmetics. Proof obligations generated from such annotations would be impossible to discharge by existing theorem provers. For practical purposes one needs instead annotations containing all the necessary information in a clear and direct manner and not obscured by Gödelization.

Therefore, in contrast to theory, all of today’s deductive verification systems presuppose certain types of additional, non-requirement annotations to be given by the user. It is neither given nor expected that an annotation-based verification system is relatively complete in the sense of Def. 6. In practice, completeness

of a verification system means that if the program is correct w.r.t. its *given* requirement specification REQ , then some auxiliary specification AUX exists allowing to prove this.

Definition 7 (Annotation completeness). *A verification system (\vdash_S, Th_S) is annotation complete if, for each program P and specification REQ with*

$$\models P+REQ ,$$

there is (a) a set AUX of annotations (not containing any assume clauses), (b) an order $<$ on the annotations, and (c) a set $Arith$ of valid arithmetical formulas such that

$$Th_S \cup Arith \vdash_S P \dot{+}_z (REQ \cup AUX) .$$

The completeness of the whole verification process depends on completeness of the components of the toolchain. As already described, the toolchain usually consists of a VCG stage and an automated theorem proving or SMT backend. The VCG must be able to generate valid formulas provided the auxiliary annotations are sufficiently strong, i.e.,

$$\text{if } \models P+REQ \text{ then } Th \models_{FOL} VCG(P+(REQ \cup AUX))$$

for some AUX . Then the TP, in its turn, must be able to prove these valid formulas:

$$Th \vdash VCG(P+(REQ \cup AUX)) .$$

The users, who serve as an oracle for finding auxiliary annotations that are strong enough to prove a given program correct are not relevant for the completeness as long as they are considered to be omniscient and always find the required annotation (provided it exists). In practice, of course, users are not omniscient. They may very well fail to find the required auxiliary annotation, which may lead to a failure in the verification process even if the verification system is complete.

Note that, if one annotation-complete system S is stronger than another annotation-complete system S' because it can automatically derive additional annotations (it may, e.g., include a generator for loop invariants), then life is easier for the user of S ; proofs will be found more often using S and with less effort (less auxiliary annotations). Nevertheless, both systems S and S' are annotation-complete; there are no different degrees of annotation completeness.

Essential and Non-essential Annotations. When a verification system is used that is annotation complete (Def. 7) but not relatively complete (Def. 6), i.e., any annotation-based verification system, then there are *essential* auxiliary annotations that cannot be omitted without losing the existence of a proof. Besides such essential annotations there are *non-essential* annotations that are not needed for the existence of a proof but for finding it more easily.

Definition 8 (Essential annotation). *Given a verification system (\vdash_S, Th_S) , a program P , a specification REQ with $\models P+REQ$, and a set AUX of annotations and an order $<$ with*

$$Th_S \cup Arith \vdash_S P+(REQ \cup AUX)$$

for some set $Arith$ of valid arithmetical formulas.

A subset $AUX_{ess} \subset AUX$ is essential if

$$Th_S \cup Arith \not\vdash_S P+(REQ \cup (AUX \setminus AUX_{ess})) .$$

Otherwise it is non-essential.

The notion of essential annotations (Def. 8) has some awkward properties, which make it difficult to recognize essential annotations in practice. It is possible that some subsets $A, A' \subset AUX$ are both essential but $A \cup A'$ is not. This happens frequently if, for example, A is needed for the proof of A' and A' is needed for the proof of A . Also, there is in general no single minimal set of essential annotations. In fact there may be completely different sets of essential auxiliary annotations for proving the same requirement that are both minimal but disjoint.

Besides the question of whether an annotation may be omitted or not, one may also be interested in the question of whether it can be replaced by a weaker annotation.

Definition 9 (Strongly essential annotation). *A subset $AUX_{ess} \subset AUX$ is strongly essential if*

$$Th_S \cup Arith \not\vdash_S P+REQ \cup ((AUX \setminus AUX_{ess}) \cup AUX')$$

for all AUX' that are weaker than AUX_{ess} , i.e., $AUX_{ess} \Rightarrow AUX'$.

While an auxiliary annotation that is strongly essential cannot be replaced by a weaker annotation, it may well be possible to replace it by an equivalent annotation that is “simpler” in some practical way not covered by Definition 9 (e.g., easier to understand for human users).

Note that even with the notion of strongly essential annotations, there is in general no single minimal set of auxiliary annotations.

It is important for a user to know which annotations are essential because during the verification process many auxiliary annotations are added. And as too many annotations clutter the program and make it harder to find a proof, users often remove unneeded annotations. This carries the danger that simple but essential annotations get removed by accident, which – as experience shows – leads to hard to solve problems in the search for a set of annotations with which a proof can be constructed. Thus, to understand which annotations may be essential, users have to possess a certain knowledge about the inner workings of a verification system. As further discussed in Section 3.4, we also suggest to enrich the annotation languages with a syntactical way (e.g., a key word) to distinguish between the two kinds of annotations.

3.3 Possible Failures in Authoring Annotations

In the following, we use a concrete example to illustrate the three different ways in which authoring annotations may fail.

Annotations and Program Code Can Be In Conflict. A program P and an annotation $SPEC$ are in conflict if the program does not fulfill the specification: $\not\models P+SPEC$.

Consider the code in Figure 1 together with the requirement to compute the minimum of a given array of length `size`. The precondition of the method (keyword `requires`) states that `array` points to a C array in memory with positive length `size`, which is not modified outside the current thread (the latter enables sequential reasoning). The post-condition of the method (keyword `ensures`) states that the result of the method is (a) less than or equal to all elements and (b) contained in the array. We assume in the following that this is the right set of requirement annotations.

One possible error that could occur in the program is that the variable `min` has never been initialized (line labeled (A) missing). The resulting program is legal C code, but depending on the random initial value of `min` and the contents of the array, may fail to compute the minimum, and it does not satisfy the annotations.

For this conflict, the VCC system is able to provide a counter-example. It demonstrates that the second loop invariant does not hold when the loop is entered. The variable assignment returned as counter-example is: `size = 1`, `min = 0`, `array[0] = 1`.

Annotations Can Be Too Weak. An auxiliary annotation AUX is too weak if $\models P+REQ \cup AUX$, i.e., the program is correct w.r.t. the specification, but this cannot be shown. There are now two cases to distinguish:

1. The VCG produces valid verification conditions, i.e.,

$$Th \models_{FOL} VCG(P+(REQ \cup AUX)) ,$$

and there is a proof for this, i.e.,

$$Th \vdash VCG(P+(REQ \cup AUX)) ,$$

but the TP stage runs out of resources before finding a proof.

2. Something essential is missing from AUX and at least one of the verification conditions generated by the VCG is invalid:

$$Th \not\models_{FOL} VCG(P+(REQ \cup AUX)) ,$$

and (because of soundness) no proof exists, that is:

$$Th \not\vdash VCG(P+(REQ \cup AUX)) .$$

```

#define uint unsigned int

int min(int *array, uint size)
  _(requires size > 0)
  _(requires \mutable_array(array, size))
  _(ensures \forallall uint i; 0<=i && i<size ==>
      result <= array[i])
  _(ensures \exists uint i; 0<=i && i<size &&
      result == array[i])
{
  uint i;
  int min;
  min = array[0]; // * (A) *
  for (i = 0; i < size; i++)
    _(invariant \forallall uint j; 0 <= j && j < i ==>
        array[j] >= min)
    _(invariant \exists uint j; 0 <= j && j < size &&
        min == array[j]) // * (B) *
    { if (array[i] < min) min = array[i]; }
  return min;
}

```

Fig. 1. Computing the smallest element of an array by simple iteration

In Case (1), no counter-example is available and the user has limited recourse – to assist the user, VCC provides tools for inspecting the duration of proof attempts for single proof obligations as well as identifying axioms that are “costly” for the prover to instantiate, leading to an inefficient proof search. In Case (2), a counter-example for the validity of the verification condition may be constructed. We give an example for this latter case.

Assume that the second loop-invariant has been forgotten (label (B) in the program in Fig. 1). Without that invariant, the system cannot verify the second post-condition. The generated counter-example is still the same as above, but this time it shows that the loop invariants (after the loop terminates) do not logically entail the post-condition.

Annotations Can Be Inadequate. An annotation is inadequate when it does not mean what its author thinks it means. Verification of inadequate annotations will thus not have the expected impact in the real world. By its very nature, user input cannot easily be verified or tested for adequacy. But, apart from many systematic approaches for elicitation of requirements (which we will not cover here), there are a number of ways in which verification technology can assist its user to formulate meaningful specifications.

First, the builders of verification systems can work on formalisms that do not make it unnecessarily hard for the users to express their exact intentions. Second, the verification systems can produce a proof or a trace to justify the result. Inspection of the proof is a very effective – if costly – measure to combat

misunderstandings in the meaning of the proof obligation. There are reports that users of verification systems monitor the prover running time to detect verification based on inadvertently inconsistent specifications (a particular case of inadequacy). In addition, VCC can check for inconsistencies in the specification by trying to prove *false* at the different execution branches of the program – this of course can also only give an indication whether the specification is consistent or not.

Third, a whole new class of sanity checks based on mutation has been developed lately for automated program verification with model checking [10]. After a successful verification attempt, the query (the program or the specification) is mutated and the deduction is repeated. If verification succeeds again, then the mutated part of the query probably plays no role in determining the outcome. This indicates a problem with the query.

3.4 Improving the Annotation Languages and Methodologies

Annotation-based verification systems are currently not designed for completeness in the sense that theorem provers are (Def. 6). They are designed for completeness in a different sense (Def. 7), requiring the user as an oracle to provide sufficient auxiliary annotations in the form of, e.g., loop invariants or assertions.

Theoretically the user could always give auxiliary annotations of maximal strength (i.e., logically entailing all other possible annotations), but this is not feasible in practice. Instead, one is interested in a weak set of auxiliary annotations that is still sufficient. Consequently, it is extremely important for the user to have knowledge about which kind of annotations are essential for the given VCG – even in cases where the requirement to be verified is comparatively simple. Without that knowledge they may continue to add the wrong annotations. It is therefore essential to provide user documentation on what kind of auxiliary annotations are needed by a verification system.

Moreover, requirement and auxiliary annotations must be syntactically distinguished. That makes specifications clearer and easier to read and understand. In certification processes it is indispensable to have a very clear understanding of which annotations form the requirement specification that has been verified.

It is preferable to keep the two in separate files: for instance, requirement annotations in the header file and auxiliary annotations in the C source file. Where no such separation is possible, keywords (in the style of visibility modifiers `public` and `private`) should be used.

4 Using Data Abstractions in Annotation-Based Verification Systems

In this section, we discuss the use of data abstractions in annotation-based verification systems, and how it can be improved. For that, we first introduce parts of the VCC methodology and the VCC annotation language only as far as needed for the examples. For a more detailed description of the VCC methodology, see [3,2].

The example we use as illustration is taken from the 1st Verified Software Competition [9]. The goal of the competition was to implement, formally specify and verify an algorithm that solves a problem defined in natural language. Our example is based upon the following requirement:

Problem: Searching a Linked List. Given a linked-list representation of a list of integers, find the index of the first element that is equal to zero. Show that the program returns a number i equal to the length of the list if there is no such element. Otherwise, the element at index i must be equal to zero, and all the preceding elements must be non-zero.

4.1 The VCC Approach

The particular solution presented here, consisting of a C implementation and a specification in the VCC language, was developed after the competition by the team “VC Crushers” (see [9])². It is an optimized version of their competition solution – the amount of auxiliary annotations is kept to a minimum sufficient to verify the requirement specification. Another VCC formalization of the list data structure that introduces a more general abstraction suited for a large range of applications is also made available by the team (because of space restrictions, we cannot include this more general but also more complex solution here).

The concrete C implementation of the list data structure and the C method `find` that is a solution to the above problem definition is shown in Fig. 2. Annotations are given in VCC syntax and are enclosed in labelled frames throughout the code.

Linked List Data Structure. In the implementation of the linked list data structure (`struct List`), the field `data` contains the value stored in a node of the list, and the field `tail` points to the rest of the list. Note that each node of the list also stores the length of its tail (including the node itself). In this implementation, the end of a list is thus not indicated by a null pointer or a sentinel node but by the value of `length` being zero.

The semantics of the `length` field as well as an abstract representation of the list’s contents are specified by the object invariants in the block labelled $\langle OI \rangle$: For each node, information about the elements of the sublist starting at that node is stored in the map `vals` in ghost state (a specification state separate from the normal C memory). The invariant I_1 defines the abstraction relation between the list and its abstraction `vals`. The abstraction from linked list to array is needed because the built-in data type `array` allows quantification and recursion over the elements of the list. A direct quantification over the elements is not possible in first-order logic because reachability is not first-order definable.

To be able to access all elements of the list (via the `data` field of the structure), each node is given exclusive ownership to the next node in the list (mine-annotation of invariant I_2). In addition, the invariant I_2 implies acyclicity of the list, as `length` is bounded and decreasing for each element that can be reached via the `tail` pointer.

² For better readability, we have slightly modified the source code of the example.

```

typedef struct List {
  int data;
  struct List *tail;
  unsigned length;
  _(ghost int vals[unsigned])
  _(invariant vals == \lambda unsigned i;
    i < length ? (i == 0 ? data
                  : tail->vals[i - 1])
    : 0
  )
  _(invariant length == 0
    || (\mine(tail) && length == tail->length + 1))
} List, *PList;

unsigned find(PList l)
{
  PList p;
  for (p = l; p->length != 0; p = p->tail)
  {
    _(requires \wrapped(l))
    _(ensures \result <= l->length)
    _(ensures \result < l->length
      ==> l->vals[\result] == 0)
    _(ensures \forall unsigned i; i < \result
      ==> l->vals[i] != 0)
  }
  for (p = l; p->length != 0; p = p->tail)
  {
    _(invariant p->length <= l->length)
    _(invariant p \in \domain(l))
    _(invariant p->vals == \lambda unsigned j;
      j < p->length
      ? l->vals[l->length - p->length + j]
      : 0)
    _(invariant \forall unsigned j;
      j < l->length - p->length ==> l->vals[j] != 0)
  }
  _(assert \forall unsigned j; j < p->tail->length
    ==> p->tail->vals[j] == p->vals[j + 1])
  if (p->data == 0) {
    break;
  }
  return l->length - p->length;
}

```

Fig. 2. Annotated C source code of find

Implementation of the `find` Algorithm. Using the C data structure `List`, the `find` method can be implemented by iterating over the list’s elements via the tail pointer in a `for`-loop.

Annotations are used within the method at three locations – namely for the method contract $\langle MC \rangle$, the loop invariant $\langle LI \rangle$ and as an auxiliary annotation inside the loop body $\langle AN \rangle$.

The method contract in block $\langle MC \rangle$ specifies that the behavior of the method conforms to its specification in natural language. The pre- and postconditions $\langle MC \rangle$, together with the invariants $\langle OI \rangle$ of the list data structure, capture the intended semantics of `find`. $\langle MC \rangle$ and $\langle OI \rangle$ constitute the methods *requirement specification*.

To be able to verify the implementation of `find` to be correct w.r.t. to its requirement specification, a set of four essential auxiliary annotations has to be provided in the form of loop invariants (block $\langle LI \rangle$).

The invariants I_3 and I_5 state that the abstraction of the “iterator” `p` is always a suffix of the abstraction of the input list `l`.

As each access to a list element `p->data` inside the loop has to be shown to be a valid access, additional justification has to be provided to VCC to be able to prove this. This justification is given with the invariant I_4 – the property depends on the fact that `p` is also a sublist of `l` in the concrete representation.

Even in this simple example, a non-essential auxiliary annotation is needed at location $\langle AN \rangle$ in order to be able to show that the loop body preserves the third loop invariant. It asserts that the `vals` fields related to two adjacent nodes are abstractions of the appropriate sublists starting at those nodes.

In order to come up with the right auxiliary annotations, in this case, the user has to know about the inner workings of VCC, respectively the strengths of the underlying SMT solver Z3.

4.2 Separation of Concerns: Annotation-Based Verification and Algebraic Specifications

In the example shown in the previous section, annotations with different purposes are intermingled. In the following, we suggest to use techniques known from abstract data type specifications to provide a clean separation of requirement and auxiliary annotations, and to make the specification more readable.

Assuming that we have defined an abstract data type `IntList`, an abstraction function `abs` from concrete linked lists to abstract lists, as well as an abstract (specification) function `absfind` on `IntList`, a good method contract for `find` could look like this:

```

unsigned find(PList l)  $\langle MC \rangle$ 
  _(requires \wrapped(l))
  _(ensures \result == absfind(abs(l)))

```

This contract is very compact and easy to understand. It simply states that `find` returns the same integer value that is the result of the abstract operation `absfind` on the abstraction of the input list.

In the VCC example presented in Sect. 4.1, the equivalent of `absfind` is implicitly given by postconditions $\langle MC \rangle$ of method `find`. The abstraction function `abs` is concealed within the definition of structure `List`, namely in the invariants in $\langle OI \rangle$.

Of course, to complete the specification, we now have to define `IntList` and `absfind`. The required syntax is not available in VCC (yet). We suggest to use a syntax for abstract data type definitions based on the Common Algebraic Specification Language (CASL) [11]. A possible definition then would look like this:

```

spec IntList =
  free type List ::= nil | cons(Int; List)
then vars i, e: Int; l, l': List
  op absfind : List -> Int
    * absfind(nil) = 0
    * absfind(cons(e, l)) = 1 when e = 0
                                else absfind(l) + 1
  ops append : List x List -> List
    tail : List -> List
    * append(nil, l) = l
    * append(cons(e, l'), l) = cons(e, append(l', l))
    * tail(nil) = nil
    * tail(cons(e, l)) = l
  within implementation find
end

```

To be able to specify the implementation of `find` with the help of the abstract data type, the data type is equipped with the externally visible operation `absfind` that captures the semantics of `find` according to the problem definition. In addition, the two operations `append` and `tail` are defined with the usual semantics.

Compared to the rudimental abstraction given by the map `vals` in Sect. 4.1, this specification is additional overhead in terms of lines of code. However, the compactness of the map specification is partly due to the fact that maps are a built-in feature of the verification tool. Furthermore, the flexibility offered by defining arbitrary abstract data types in our opinion clearly outweighs the annotation overhead, as we can choose the abstract data type representation that matches the implementation data types best. Lastly, the above definition is to a large extent reusable and can be seen as a sort of library definition.

To relate the concrete implementation data type `List` to its abstract data type counterpart `IntList`, we have two options: (A) defining an abstraction function `abs` from `List` to `IntList` (as already mentioned above), and (B) axiomatizing the abstraction using concrete implementations for the (abstract) constructors `nil` and `cons`.

For option (A), the abstraction function `abs` is defined in terms of the concrete implementation details (i.e., field `data` and pointer `tail`):

```

                                                                    ⟨OI⟩
_ (spec IntList abs (PList l)
  returns (l->length == 0 ? nil :
           cons(l->data, abs(l->tail)))
)

```

With this definition, one of our goals is already achieved, namely providing a clearly differentiated and discernible specification construct that couples the concrete and abstract data types. However, this definition does not hide the implementation details of the linked list data structure from callers of the `find` method.

The alternative solution (B) uses concrete implementations of the constructors of the list data structures (not shown in this paper). Then, no explicit definition of the abstraction function as in (A) is needed. The relation between the abstract and the concrete constructors can, for example, be specified as follows:

```

                                                                    ⟨OI⟩
PList cons(int e, PList l)
  _ (requires \wrapped(l))
  _ (ensures abs(\result) == cons(e, abs(l)) )

```

Note that the above method contract of `cons` does not use any implementation details of the linked list data type in C, so that these details do not become part of the requirement specification.

Regardless of which alternative (A) or (B) is chosen, due to the separation of abstract and concrete representation of the list data type, the annotation overhead of the `List` data structure can be reduced:

```

typedef struct List {
  int data;
  struct List *tail;
  unsigned length;
  _ (invariant \exists IntList l; abs(this) == l
     && (abs(this) == nil || \mine(tail)))
} List, *PList;

```

Only one invariant of `list` remains that is concerned with the state of the structure according to the VCC methodology (keyword `mine`), as well as enforcing the existence of an abstract element corresponding to the concrete instance of the list (which rules out cyclic linked lists).

Using the two “library” functions `tail` and `append`, almost all auxiliary annotations of our example can be simplified – for the loop invariants at location ⟨LI⟩ the new annotations are:

```

_(invariant p \in \domain(l))
_(invariant \exists IntList front;
  append(front, abs(p)) == abs(l)
  && absfind(f) == 0)

```

Furthermore, the single non-essential annotation at location $\langle LI \rangle$ becomes:

```

_(assert abs(p->tail) == tail(abs(p)))

```

5 Conclusions and Future Work

We have described and analyzed problematic properties of current annotation-based specification practices. A large part of the problem is the non-discriminatory use of the same language to specify both requirements and auxiliary (non-requirement) annotations. Among the latter some are (only) needed for efficiency of proof search, while others are essential for completeness, i.e., indispensable for proof construction. In practice, we often encountered confusion as to what the important notion of completeness means in the framework of verifying compilers. We have provided a clarification in Section 3.2. We have recommended measures to alleviate the specification bottleneck in Sections 3.4 and 4.2. Beyond that, we see the following issues as worth further exploration.

The distinction between requirement and auxiliary annotations is always relative to a module boundary. Some aspects of modularity are imposed by the programming language, while others have to be defined by the specification language and verification calculus. The designers of the latter should make their modularity concepts (syntactically) explicit and better educate the users about them. A hiding operator for annotations may be appropriate when composing modules.

It is important to keep related parts of specifications together and not mix different levels of abstraction. This tenet is frequently violated by the widespread practice of mixing ghost code (providing an abstract specification) and real code (implementing it) – often on the level of individual statements. While a separation is desirable, it is not yet clear how to disentangle the two.

The verification system should track (and disclose) dependencies between annotations. All annotations should carry a unique textual identifier (label). For any given annotation, the user must be informed about which other annotations are necessary to prove it. One way to accomplish this is by automated deduction on the part of the system. Another way is to demand that the user explicitly attach to each annotation a set of labels naming other annotations that are to be considered as premisses in the proof.

References

1. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# programming system: An overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 49–69. Springer, Heidelberg (2005)

2. Beckert, B., Moskal, M.: Deductive verification of system software in the Verisoft XT project. In: KI 2009, Online first version available at SpringerLink (2009)
3. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: A practical system for verifying concurrent C. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 23–42. Springer, Heidelberg (2009)
4. Cook, S.A.: Soundness and completeness of an axiom system for program verification. *SIAM Journal of Computing* 7(1), 70–90 (1978)
5. 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)
6. DeLine, R., Leino, K.R.M.: BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research (2005)
7. Filliâtre, J.-C., Marché, C.: Multi-prover verification of C programs. In: Davies, J., Schulte, W., Barnett, M. (eds.) ICFEM 2004. LNCS, vol. 3308, pp. 15–29. Springer, Heidelberg (2004)
8. Harel, D.: First-Order Dynamic Logic. LNCS, vol. 68. Springer, Heidelberg (1979)
9. Klebanov, V., Müller, P., Shankar, N., Leavens, G.T., Wüstholtz, V., Alkassar, E., Arthan, R., Bronish, D., Chapman, R., Cohen, E., Hillebrand, M., Jacobs, B., Leino, K.R.M., Monahan, R., Piessens, F., Polikarpova, N., Ridge, T., Smans, J., Tobies, S., Tuerk, T., Ulbrich, M., Weiß, B.: The 1st verified software competition: Experience report. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 154–168. Springer, Heidelberg (2011) Materials available at, www.vscomp.org
10. Kupferman, O.: Sanity checks in formal verification. In: Baier, C., Hermanns, H. (eds.) CONCUR 2006. LNCS, vol. 4137, pp. 37–51. Springer, Heidelberg (2006)
11. Mosses, P.D. (ed.): CASL Reference Manual – The Complete Documentation of the Common Algebraic Specification Language. LNCS, vol. 2960. Springer, Heidelberg (2004)
12. Schulte, W., Songtao, X., Smans, J., Piessens, F.: A glimpse of a verifying C compiler. In: Proceedings, C/C++ Verification Workshop (2007)
13. Zeller, A.: Mining specifications: A roadmap. In: Proceedings, The Future of Software Engineering, Zurich, Switzerland, pp. 173–182. Springer, Heidelberg (2010)

Program Specialization via a Software Verification Tool*

Richard Bubel, Reiner Hähnle, and Ran Ji

Department of Computer Science and Engineering
Chalmers University of Technology, 41296 Gothenburg, Sweden
{bubel,reiner,ran.ji}@chalmers.se

Abstract. Partial evaluation is a program specialization technique that allows to optimize a program for which partial input is known. We propose a new approach to generate specialized programs for a Java-like language via the software verification tool KeY. This is achieved by symbolically executing source programs interleaved with calls to a simple partial evaluator. In a second phase the specialized programs are synthesized from the symbolic execution tree. The correctness of this approach is guaranteed by a bisimulation relation on the source and specialized programs.

1 Introduction

Symbolic execution [13] and partial evaluation [12] are both generalizations of standard interpretation of programs in different ways: while symbolic execution permits interpretation of a program with symbolic (i.e., unspecified) initial values, the aim of partial evaluation is to transform a program with partially specified input values into a (hopefully, more efficient) program that has only the unspecified arguments as input. For fully specified input arguments the result of both mechanisms is standard program interpretation.

Our previous work [5] showed how to speed up the symbolic execution engine by interleaving with partial evaluation. On the other hand, an important question that can be asked is the possibility of achieving a more sophisticated program specializer via symbolic execution interleaved with simple partial evaluation operations. Another interesting question is whether the specialized program will behave the same as the source program with respect to the observable output, i.e., the soundness of the program specialization procedure. This paper tries to give an answer.

We propose a new approach to specialize Java-like programs via the software verification tool KeY, in which a symbolic execution engine is used. It is a two-phase procedure that first symbolically executes the program interleaved with

* This work has been partially supported by the EU project FP7-ICT-2007-3 HATS *Highly Adaptable and Trustworthy Software using Formal Models* and the EU COST Action IC0701 *Formal Verification of Object-Oriented Software*.

a simple partial evaluator, and then synthesizes the specialized program in the second phase. The soundness of the approach is proved.

The paper is organized as follows: In Sect. 2 we introduce a Java-like programming language PL as the working language in this paper and define its program logic. Sect. 3 presents the sequent calculus for PL. In Sect. 4 we integrate a simple partial evaluator into the symbolic execution engine. In Sect. 5 we introduce the bisimulation modality and define its sequent calculus. Sect. 6 shows how to generate specialized programs using our approach. Sect. 7 discusses related work. Sect. 8 concludes and addresses future work.

2 Dynamic Logic

Dynamic logic (DL) [9] is a representative of multi-modal logic tailored towards program verification. The programs to be verified against their specification occur in unencoded form as first class citizens of dynamic logic similar to Hoare logic [10] and avoid the encoding of programs.

Sorted first-order dynamic logic is sorted first-order logic plus two additional kinds of modalities: $[\cdot]$ (box) and $\langle \cdot \rangle$ (diamond). The first parameter takes a program and the second parameter a dynamic logic formula. Let p denote a program and ϕ a dynamic logic formula then

- $[p]\phi$ is a DL-formula and, informally, expresses that if p is executed and terminates *then* in all reached final states ϕ holds;
- $\langle p \rangle \phi$ is a DL-formula. Informally, it means that if p is executed then it terminates *and* in at least one of the reached final states ϕ holds.

We consider from now on only deterministic programs: Hence, a program p executed in a given state s *either* terminates and reaches exactly *one* final state *or* it does not terminate and there are no final states reachable from s upon execution of program p . In this setting, the box modality expresses *partial correctness* of a program, while the diamond modality coincides with *total correctness*. With the above statements we can see that Hoare logic is subsumed by dynamic logic. The Hoare triple $\{pre\} p \{post\}$ can be expressed as the DL formula $pre \rightarrow [p]post$.

In the remainder of this section we introduce some basic concepts of dynamic logic. We follow thereby closely the KeY-approach [3].

Programming Language. We consider a simple Java-like programming language called PL. It provides features like interfaces, classes, attributes, method polymorphism (but not method overloading). PL does not support multi-threading, floating points or garbage collection. For ease of presentation we omit also class and object initialization, exceptions as well as `break` or `continue` statements and require non-void methods to have a single point of return at the end of the method. Further, the only supported primitive types are `boolean` and `int`.

Fig. 1 shows a PL program which we use as a running example throughout the paper.

```

public class OnLineShopping {
    boolean cpn;
    public int read() { /* read price of item */ }
    public int sum(int n) {
        int i = 0;
        int count = n;
        int tot = 0;
        while(i <= count) {
            int m = read();
            if(i >=2 && cpn) { tot = tot + m * 9 / 10; i++; }
            else { tot = tot + m; i++; }
        }
        return tot;
    }
}

```

Fig. 1. Example PL program

This PL program could be used in an online shopping session. The `read()` method collects the price for each item. The `sum()` method calculates the total amount to be paid when purchasing n items. If the customer provides a coupon and purchases at least 2 items, then a 10% discount will apply from the second item onwards.

Dynamic Logic. Given a PL program \mathcal{C} including interface and class declarations. Program \mathcal{C} is in the following referred to as *context program*. We define our dynamic logic PL-DL(\mathcal{C}) as follows:

Definition 1 (PL-Signature $\Sigma_{\mathcal{C}}$). *The signature $\Sigma_{\mathcal{C}} = (S, \preceq, \text{Pred}, \text{Func}, \text{LVar})$ consists of*

- a set of names S called sorts containing at least one sort for each primitive type and for each interface I and class C declared in \mathcal{C} as well as the `Null` sort:

$$S \supseteq \{\text{int}, \text{boolean}, \text{Null}\} \cup \{T \mid f.a. \text{ interfaces or classes } T \text{ declared in } \mathcal{C}\}$$

- The partial subtyping order $\preceq: S \times S$ models the subtype hierarchy of \mathcal{C} faithfully.
- An infinite set of function symbols $\text{Func} := \{f : T_1 \times \dots \times T_n \rightarrow T \mid T_i, T \in S\}$. We call $\alpha(f) = T_1 \times \dots \times T_n \rightarrow T$ the arity of the function symbol. $\text{Func} := \text{Func}_r \cup \text{PV} \cup \text{Attr}$ is further divided into disjoint subsets:
 - Func_r rigid function symbols,
 - PV program variables i, j (non-rigid constants), and
 - attribute function symbols Attr , where for each attribute a of type T declared in class C an attribute function $a@C : C \rightarrow T \in \text{Attr}$ exists.

We omit the `@C` from attribute names if no ambiguity arises.

- An infinite set of predicate symbols $\text{Pred} := \{p : T_1 \times \dots \times T_n \mid T_i \in S\}$.
- A set of logical variables $\text{LVar} := \{x : T \mid T \in S\}$.

Terms and formulas are defined as usual. The grammar below together with the canonical well-typedness conditions defines the syntax:

$$\begin{aligned}
t &::= x \mid \mathbf{i} \mid t.\mathbf{a} \mid f(t, \dots, t) \mid (\phi ? t : t) \mid \\
&\quad \mathbb{Z} \mid \text{TRUE} \mid \text{FALSE} \mid \text{null} \mid \{u\}t \\
u &::= \mathbf{i} := t \mid t.\mathbf{a} := t \mid u \parallel u \\
\phi &::= \neg\phi \mid \phi \circ \phi \ (\circ \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}) \mid (\phi ? \phi : \phi) \mid \\
&\quad \forall x : T.\phi \mid \exists x : T.\phi \mid [\mathbf{p}]\phi \mid \langle \mathbf{p} \rangle \phi \mid \{u\}\phi
\end{aligned}$$

where $\mathbf{a} \in \text{Attr}$, $f \in \text{Func}$, $\mathbf{i} \in \text{PV}$, $x : T \in \text{LVar}$, and \mathbf{p} is a sequence of executable statements in PL. The elements of category u are called *updates* and used to describe state changes. An *elementary update* $\mathbf{i} := t$ or $t.\mathbf{a} := t$ is a pair of location and term. Its intended meaning is that of an assignment. Updates applied on terms or formulas are again terms or formulas. They can be composed to *parallel updates* playing then the role of simultaneous assignments.

In the remaining paper, we use the notion of a program to refer to a sequence of executable PL-statements. If we want to include class, interface or method declarations, we either include them explicitly or make a reference to the context program \mathcal{C} .

The next step is to assign meaning to PL-DL terms and formulas. A formula in dynamic logic is evaluated with respect to a Kripke structure, in our case a PL-DL Kripke structure:

Definition 2 (Kripke structure $\mathcal{K}_{\Sigma_{PL}}$). A PL-DL Kripke structure is a triple $\mathcal{K}_{\Sigma_{PL}} = (\mathcal{D}, I, St)$ with

- a set of elements \mathcal{D} called domain,
- an interpretation I with
 - $I(s) = \mathcal{D}_s$, $s \in S$ assigning each sort its non-empty domain \mathcal{D}_s . It adheres to the restrictions imposed by the subtype order \preceq ; **Null** is always interpreted as a singleton set and subtype of all reference (class and interface) types.
 - $I(f) : \mathcal{D}_{T_1} \times \dots \times \mathcal{D}_{T_n} \rightarrow \mathcal{D}_T$ for each rigid function symbol $f : T_1 \times \dots \times T_n \rightarrow T \in \text{Func}_r$.
 - $I(p) \subseteq \mathcal{D}_{T_1} \times \dots \times \mathcal{D}_{T_n}$ for each rigid predicate symbol $p : T_1 \times \dots \times T_n \in \text{Pred}$.
- a set of states St assigning non-rigid function symbols a meaning: Let $s \in St$ then $s(\mathbf{a}@C) : \mathcal{D}_C \rightarrow \mathcal{D}_T$, $\mathbf{a}@C \in \text{Attr}$ and $s(\mathbf{i}) : \mathcal{D}_T$, $\mathbf{i} \in \text{PV}$.

The pair $D = (\mathcal{D}, I)$ is called a first-order structure.

Finally, a *variable assignment* $\beta : \text{LVar} \rightarrow \mathcal{D}_T$ maps a logic variable $x : T$ to its domain \mathcal{D}_T . An update, program, term or formula ξ is evaluated with respect to a given first-order structure $D = (\mathcal{D}, I)$, a state $s \in St$ and a variable assignment β as $val_{D, I, s, \beta}(\xi)$. The evaluation function val is defined recursively on the term and formula structure. Fig. 2 shows an excerpt of its definition.

Some words on the semantics of updates. While elementary updates have the same meaning as assignments, the semantics of parallel updates is slightly more complicated. We explain them by example:

$$\begin{aligned}
val_{D,s,\beta}(x) &= \beta(x), x \in \text{LVar} \\
val_{D,s,\beta}(f(t_1, \dots, t_n)) &= D(f)(val_{D,s,\beta}(t_1), \dots, val_{D,s,\beta}(t_n)) \\
val_{D,s,\beta}(\mathbf{x}) &= s(\mathbf{x}), \mathbf{x} \in \text{PV} \\
val_{D,s,\beta}(o.\mathbf{a}) &= s(\mathbf{a})(val_{D,s,\beta}(o)), \mathbf{a} \in \text{Attr} \\
val_{D,s,\beta}(\neg\phi) &= tt \text{ iff } val_{D,s,\beta}(\phi) = ff \\
val_{D,s,\beta}(\psi \wedge \phi) &= tt \text{ iff } val_{D,s,\beta}(\psi) = tt \text{ and } val_{D,s,\beta}(\phi) = tt \\
val_{D,s,\beta}(\psi \vee \phi) &= tt \text{ iff } val_{D,s,\beta}(\psi) = tt \text{ or } val_{D,s,\beta}(\phi) = tt \\
val_{D,s,\beta}(\psi \rightarrow \phi) &= val_{D,s,\beta}(\neg\psi \vee \phi) \\
\\
val_{D,s,\beta}((\psi ? \xi_1 : \xi_2)) &= \begin{cases} val_{D,s,\beta}(\xi_1) & \text{if } val_{D,s,\beta}(\psi) \\ val_{D,s,\beta}(\xi_2) & \text{otherwise} \end{cases} \\
val_{D,s,\beta}(\mathbf{x} := v) &= s', \text{ with } \begin{cases} s'(x) = val_{D,s,\beta}(v) \\ s'(y) = s(y) & y \neq x \end{cases} \\
val_{D,s,\beta}(o.\mathbf{a} := v) &= \{s'\}, s = s' \text{ except } s'(a)(val_{D,s,\beta}(o)) = val_{D,s,\beta}(v) \\
val_{D,s,\beta}([s_1; s_2]\phi) &= \begin{cases} val_{D,s',\beta}([s_2]\phi), \{s'\} = val_{D,s,\beta}(s_1) \\ tt, s_1 \uparrow \end{cases} \\
val_{D,s,\beta}(\{\text{if}(e) \{p\} \text{ else } \{q\}\}\phi) &= val_{D,s,\beta}(\{\text{T } \mathbf{b}; \mathbf{b} = \mathbf{e}; \text{if}(e) \{p\} \text{ else } \{q\}\}\phi) \\
val_{D,s,\beta}(\{\text{if}(\mathbf{b}) \{p\} \text{ else } \{q\}\}\phi) &= \begin{cases} val_{D,s,\beta}(\{p\}\phi), s(\mathbf{b}) = val_{D,s,\beta}(\text{TRUE}) \\ val_{D,s,\beta}(\{q\}\phi), \text{ otherwise} \end{cases} \quad (\mathbf{b} \in \text{PV})
\end{aligned}$$

Fig. 2. Definition of evaluation function val (excerpt)

Example 1 (Update semantics)

- Evaluating $\{\mathbf{i} := \mathbf{j} + 1\} \mathbf{i} \geq \mathbf{j}$ in a state s is identical to evaluate the formula $\mathbf{i} \geq \mathbf{j}$ in a state s' which coincides with s except for the value of \mathbf{i} which is evaluated to the value of $val_{D,s,\beta}(\mathbf{j} + 1)$.
- Evaluation of the parallel update $\mathbf{i} := \mathbf{j} \parallel \mathbf{j} := \mathbf{i}$ in a state s leads to the successor state s' identical to s except that the values of \mathbf{i} and \mathbf{j} are swapped.
- The parallel update $\{\mathbf{i} := 3 \parallel \mathbf{i} := 4\}$ has a conflict as \mathbf{i} is assigned different values. In such a case the conflict is resolved by using a last-one-wins semantics. Last-one-wins semantics means that the textually last occurring assignment overrides all previous ones of the same location.

We conclude the presentation of PL-DL by defining the notions of satisfiability, model and validity.

Definition 3 (Satisfiability, model and validity). *A formula ϕ*

- *is called satisfiable if there exists a first-order structure D , a state $s \in St$ and a variable assignment β with $val_{D,s,\beta}(\phi) = tt$ (short: $D, s, \beta \models \phi$).*
- *has a model if there exists a first-order structure D , a state $s \in St$, such that for all variable assignments β : $val_{D,s,\beta}(\phi) = tt$ holds (short: $D, s \models \phi$).*
- *is valid if for all first-order structures D , states $s \in St$ and for all variable assignments β : $val_{D,s,\beta}(\phi) = tt$ holds (short: $\models \phi$).*

3 Sequent Calculus

To analyze a PL-DL formula for validity, we use a Gentzen style sequent calculus. A sequent

$$\underbrace{\phi_1, \dots, \phi_n}_{\Gamma} \Rightarrow \underbrace{\psi_1, \dots, \psi_m}_{\Delta}$$

is a pair of sets of formulas Γ (antecedent) and Δ (succedent). Its meaning is identical to the meaning of the formula

$$\bigwedge_{\phi \in \Gamma} \phi \rightarrow \bigvee_{\psi \in \Delta} \psi$$

A sequent calculus rule

$$\text{rule} \frac{\overbrace{\Gamma_1 \Rightarrow \Delta_1 \quad \dots \quad \Gamma_n \Rightarrow \Delta_n}^{\text{premises}}}{\underbrace{\Gamma \Rightarrow \Delta}_{\text{conclusion}}}$$

consists of one conclusion and possibly many premises. One example of a sequent calculus rule is the rule **andRight**:

$$\text{andRight} \frac{\Gamma \Rightarrow \phi, \Delta \quad \Gamma \Rightarrow \psi, \Delta}{\Gamma \Rightarrow \phi \wedge \psi, \Delta}$$

We call ϕ and ψ (formula) schema variables which match here any arbitrary formula. A rule is applied on a sequent s by matching its conclusion against s . The instantiated premises are then added as children of s . For example, when applying **andRight** to the sequent $\Rightarrow i \geq 0 \wedge \neg o.a = \text{null}$ we instantiate ϕ with $i \geq 0$ and ψ with $\neg o.a = \text{null}$. The instantiated sequents are then added as children to the sequent and the resulting partial proof tree becomes:

$$\frac{\Rightarrow i \geq 0 \quad \Rightarrow \neg o.a = \text{null}}{\Rightarrow i \geq 0 \wedge \neg o.a = \text{null}}$$

Fig. 3 shows a selection of first-order sequent calculus rules. A proof of the validity of a formula ϕ in a sequent calculus is a tree where

- each node is annotated with a sequent,
- the root is labeled with $\Rightarrow \phi$,
- for each inner node n : there is a sequent rule whose conclusion matches the sequent of n and there is a bijection between the rule’s premises and the children of n , and,
- the last rule application on each branch is the application of a close rule (axiom).

Axioms and Propositional Rules

$$\begin{array}{ccc}
\text{close} \frac{*}{\phi \Rightarrow \phi} & \text{closeTrue} \frac{*}{\Rightarrow \text{true}} & \text{closeFalse} \frac{*}{\text{false} \Rightarrow} \\
\text{andLeft} \frac{\Gamma, \psi, \phi \Rightarrow \Delta}{\Gamma, \phi \wedge \psi \Rightarrow \Delta} & \text{orRight} \frac{\Gamma \Rightarrow \phi, \psi, \Delta}{\Gamma \Rightarrow \phi \vee \psi, \Delta} & \text{impRight} \frac{\Gamma, \phi \Rightarrow \psi, \Delta}{\Gamma \Rightarrow \phi \rightarrow \psi, \Delta} \\
\text{andRight} \frac{\Gamma \Rightarrow \phi, \Delta \quad \Gamma \Rightarrow \psi, \Delta}{\Gamma \Rightarrow \phi \wedge \psi, \Delta} & \text{orLeft} \frac{\Gamma, \phi \Rightarrow \Delta \quad \Gamma, \psi \Rightarrow \Delta}{\Gamma, \phi \vee \psi, \Delta \Rightarrow} &
\end{array}$$

First-Order Rules

$$\begin{array}{cc}
\text{allLeft} \frac{\Gamma, \phi[x/t] \Rightarrow \Delta}{\Gamma, \forall x : T. \phi \Rightarrow \Delta} & \text{exRight} \frac{\Gamma \Rightarrow \phi[x/t], \Delta}{\Gamma \Rightarrow \exists x : T. \phi \Delta} \\
\text{allRight} \frac{\Gamma \Rightarrow \phi[x/c], \Delta}{\Gamma \Rightarrow \forall x : T. \phi, \Delta} & \text{exLeft} \frac{\Gamma, \phi[x/c] \Rightarrow \Delta}{\Gamma, \exists x : T. \phi \Rightarrow \Delta} \\
& c \text{ new, freeVars}(\phi) = \emptyset
\end{array}$$

Fig. 3. First-order calculus rules (excerpt)

So far the considered rules were pure first-order reasoning rules. The calculus design regarding rules for formulas with programs is discussed next. We consider only the box modality variant of these rules.

Our sequent calculus variant is designed to stepwise symbolically execute a program. It behaves for most parts as a symbolic program interpreter. Symbolic execution as a means for program verification goes back to King [L3]. Symbolic execution means that upon program execution the initial values of the input variables, fields etc., are symbolic values (terms) instead of concrete ones. The program then performs algebraic computations on those terms instead of actually computing concrete values.

We explain the core concepts along a few selected rules. Starting with the assignment rule:

$$\text{assignLocalVariable} \frac{\Gamma \Rightarrow \{\mathcal{U}\}\{x := \text{litVar}\}[\omega]\phi, \Delta}{\Gamma \Rightarrow \{\mathcal{U}\}[x = \text{litVar}; \omega]\phi, \Delta}$$

where $x \in \text{PV}$, and litVar is either a boolean/integer literal or a program variable, and ω the rest of the program. The assignment rule works as most program rules on the first active statement ignoring the rest of the program (collapsed into ω). Its effect is the movement of the elementary program assignment into an update.

The assignment rule for an elementary addition is similar and looks like

$$\text{assignLocalVariable} \frac{\Gamma \Rightarrow \{\mathcal{U}\}\{x := \text{litVar1} + \text{litVar2}\}[\omega]\phi, \Delta}{\Gamma \Rightarrow \{\mathcal{U}\}[x = \text{litVar1} + \text{litVar2}; \omega]\phi, \Delta}$$

There is a number of other assignment rules for the different program expressions. All of the assignment rules have in common that they operate on elementary (pure) expressions. This is necessary to reduce the number of rules and also as expressions may have side-effects that need to be “computed” first. Our calculus works in two phases: first complex statements and expressions are decomposed into a sequence of simpler statements, then they are moved to an assignment or are handled by other kinds of rules (e.g., a loop invariant rule). The decomposition phase consist mostly of so called unfolding rules such as:

unfoldAssignmentAddition

$$\frac{\Gamma \Rightarrow \{\mathcal{U}\}[\text{int } v1 = exp1; \text{int } v2 = exp2; \mathbf{x} = v1 + v2; \omega] \phi, \Delta}{\Gamma \Rightarrow \{\mathcal{U}\}[\mathbf{x} = exp1 + exp2; \omega] \phi, \Delta}$$

where $exp1, exp2$ are arbitrary (nested) expressions and $v1, v2$ new program variables not yet used in the proof or in ω .

The conditional rule is a typical representative of a program rule to show how splits in control flows are treated:

conditionalSplit

$$\frac{\Gamma, \{\mathcal{U}\} \mathbf{b} = \text{TRUE} \Rightarrow \{\mathcal{U}\}[\mathbf{p}; \omega] \phi, \Delta \quad \Gamma, \{\mathcal{U}\} \neg \mathbf{b} = \text{TRUE} \Rightarrow \{\mathcal{U}\}[\mathbf{q}; \omega] \phi, \Delta}{\Gamma \Rightarrow \{\mathcal{U}\}[\text{if } (\mathbf{b}) \{ \mathbf{p} \} \text{ else } \{ \mathbf{q} \} \omega] \phi, \Delta}$$

where \mathbf{b} is a program variable.

The calculus provides two different kinds of rules to treat loops. The first one realizes—as one would expect from a program interpreter—a simple unwinding of the loop:

$$\text{loopUnwind} \frac{\Gamma \Rightarrow \{\mathcal{U}\}[\text{if } (\mathbf{b}) \{ \bar{\mathbf{p}}; \text{while } (\mathbf{b}) \{ \mathbf{p} \} \} \omega] \phi, \Delta}{\Gamma \Rightarrow \{\mathcal{U}\}[\text{while } (\mathbf{b}) \{ \mathbf{p} \} \omega] \phi, \Delta}$$

where $\bar{\mathbf{p}}$ is identical to \mathbf{p} except for renaming of the newly declared variables in \mathbf{p} to avoid name collisions.

The major drawback of the rule is that except for cases where the loop has a fixed and known number of iterations, the rule can be applied arbitrarily often. Instead of unwinding the loop, one often used alternative is the loop invariant rule `whileInv`:

whileInv

$$\frac{\begin{array}{l} \Gamma \Rightarrow \{\mathcal{U}\} inv, \Delta \quad \text{(init)} \\ \Gamma, \{\mathcal{U}\} \{ \mathcal{V}_{mod} \} (\mathbf{b} = \text{TRUE} \wedge inv) \Rightarrow \{\mathcal{U}\} \{ \mathcal{V}_{mod} \} [\mathbf{p}] inv, \Delta \quad \text{(preserves)} \\ \Gamma, \{\mathcal{U}\} \{ \mathcal{V}_{mod} \} (\mathbf{b} = \text{FALSE} \wedge inv) \Rightarrow \{\mathcal{U}\} \{ \mathcal{V}_{mod} \} [\omega] \phi, \Delta \quad \text{(use case)} \end{array}}{\Gamma \Rightarrow \{\mathcal{U}\}[\text{while } (\mathbf{b}) \{ \mathbf{p} \} \omega] \phi, \Delta}$$

The loop invariant rule requires the user to provide a sufficiently strong formula inv capturing the functionality of the loop. The formula needs to be valid before the loop is executed (init branch) and must not be invalidated by any loop iteration started from a state satisfying the loop condition (preserves branch).

Finally, in the third branch the symbolic execution continues with the remaining program after the loop.

The *anonymizing* update \mathcal{V}_{mod} requires further explanation: We have to show that *inv* is preserved by an arbitrary iteration of the loop body as long as the loop condition is satisfied. But in an arbitrary iteration, values of program variables may have changed and outdated the information provided by Γ, Δ and \mathcal{U} . In traditional loop invariant rules, this context information is removed completely and the still valid portions have to be added to the invariant formula *inv*. We use the approach described in [3] and avoid to invalidate all previous knowledge. For this we require the user to provide a superset of all locations *mod* that are potentially changed by the loop. The anonymizing update \mathcal{V}_{mod} erases all knowledge about these locations by setting them to a fixed, but unknown value. An overapproximation of *mod* can be computed automatically.

The last rule we want to introduce is about method contracts and it is a necessity to achieve modularity in program verification. More important for this paper is that it allows to achieve a modular program specializer. Given a method $T \text{ m}(T \text{ param}_1, \dots, T_n \text{ param}_n)$ and a method contract

$$C(m) = (\text{pre}(\text{param}_1, \dots, \text{param}_n), \text{post}(\text{param}_1, \dots, \text{param}_n, \text{res}), \text{mod})$$

The formulas *pre* and *post* are the precondition and postcondition of the method with access to the parameters and to the result variable *res* (the latter only in *post*). The location set *mod* describes the locations (fields) that may be changed by the method. When we encounter a method invocation, the calculus first unfolds all method arguments. After that the method contract rule is applicable:

methodContract

$$\frac{\begin{array}{l} \Gamma \Rightarrow \{\mathcal{U}\}\{\text{param}_1 := \mathbf{v1} \parallel \dots \parallel \text{param}_n := \mathbf{vn}\}\text{pre}, \Delta \\ \Gamma \Rightarrow \{\mathcal{U}\}\{\text{param}_1 := \mathbf{v1} \parallel \dots \parallel \text{param}_n := \mathbf{vn}\}\{\mathcal{V}_{mod}\}(\text{post} \rightarrow [\mathbf{r} = \text{res}; \omega]\phi), \Delta \end{array}}{\Gamma \Rightarrow \{\mathcal{U}\}[\mathbf{r} = \mathbf{m}(\mathbf{v1}, \dots, \mathbf{vn}); \omega]\phi, \Delta}$$

In the first branch we have to show that the precondition of the method is satisfied. The second branch then allows us to assume that the postcondition is valid and we can continue to symbolically execute the remaining program. The anonymizing update \mathcal{V}_{mod} erases again all information about the locations that may have been changed by the method. About the values of these locations, the information encoded in the postcondition is the only knowledge that is available and on which we can rely in the remaining proof.

Definition 4 (Soundness). *A rule is sound if and only if the validity of the premises implies the validity of the conclusion.*

Theorem 1. *The sequent calculus rules for PL-DL are sound.*

4 Integrated Simple Partial Evaluator

In this section, we show how to integrate a simple partial evaluator into the symbolic execution engine to perform some basic partial evaluation when

symbolically executing the program. The operations defined here were also introduced in our previous paper [5], where we showed how to speed up the symbolic execution engine by interleaving these partial evaluation operations.

The basic idea is to introduce a partial evaluation operator $p \downarrow (\mathcal{U}, \phi)$ that can be attached to any program statement or expression p . The partial evaluation operator then specializes the program construct p with respect to the knowledge accumulated in update \mathcal{U} and formula ϕ . The integration is achieved by introducing special sequent calculus rules that trigger the specialization on the program under consideration.

Specialization Operator Propagation. The specialization operator needs to be propagated along the program as most of the different specialization operations work locally on single statements or expressions. During propagation of the operator, its knowledge base, the pair (\mathcal{U}, ϕ) , needs to be updated by additional knowledge learned from executed statements or by erasing invalid knowledge about variables altered by the previous statement. Propagation of the specialization operator as well as updating the knowledge base is realized by the following rewrite rule

$$(p; q) \downarrow (\mathcal{U}, \phi) \rightsquigarrow p \downarrow (\mathcal{U}, \phi); q \downarrow (\mathcal{U}', \phi')$$

This rule is sound under a number of restrictions of \mathcal{U}' , ϕ' , see [5] for details.

Constant propagation and constant expression evaluation. Constant propagation entails that if the value of a variable v is known to have a constant value c within a certain program region (typically, until the variable is potentially reassigned) then usages of v can be replaced by c . Note that c could also be another variable and in this case usages of v can be replaced by c until v is reassigned or c is changed. The rewrite rule $(v) \downarrow (\mathcal{U}, \varphi) \rightsquigarrow c$ models the replacement operation. To ensure soundness the rather obvious condition $\mathcal{U}(\varphi \rightarrow v \doteq c)$ has to be proved where c is a rigid constant (within the given region). The above rule can be easily modified to include constant expression evaluation.

For example, in Fig. 11, when executing `int count = n`; in method `sum()`, `count` will be replaced by `n` in the loop guard because both `count` and `n` are unchanged, therefore, the loop guard becomes `i <= n`. However, `int i = 0`; could not propagate `0` into the loop guard since `i` could potentially be reassigned in the loop. One interesting point is that, if we unwind the loop once according to `loopUnwind` rule introduced in Section 3, it will become `if(i <= n) ... if(i >= 2 && cpn) ... while(i <= n) ...`, where `i` is ok to be replaced by `0` in both conditional guards (because `i` is not reassigned), but not in the loop guard. The result looks like `if(0 <= n) ... if(0 >= 2 && cpn) ... while(i <= n) ...`

Dead-Code Elimination. Constant propagation and constant expression evaluation often result in specializations where the guard of a conditional (or loop) becomes constant. In this case, unreachable code in the current state and under the current path condition can be easily located and pruned. A typical

example for a specialization operation eliminating an infeasible symbolic execution branch is the rule

$$(\text{if } (b) \{p\} \text{ else } \{q\}) \downarrow (\mathcal{U}, \phi) \quad \rightsquigarrow \quad p \downarrow (\mathcal{U}, \phi)$$

which eliminates the `else` branch of a conditional if the guard can be proved true. The soundness condition of the rule is straightforward and self-explaining: $\mathcal{U}(\phi \rightarrow b \doteq \text{TRUE})$.

Continuing the example above, we can perform further specialization with the dead-code elimination rule. Since in the second conditional guard $0 >= 2$ is evaluated to *false*, the `then`-branch is pruned. The result is `if(0 <= n) int m = read(); tot = tot + m; i++; while(i <= n)`

Some other partial evaluation operations such as *Safe Field Access* and *Type Inference* are also integrated. Please refer to [5] for more details.

5 A Sequent Calculus for Bisimulation

In the previous sections we introduced a dynamic logic based on symbolic execution. In section 4 we reported about our previous work on speeding up symbolic execution by interleaving the symbolic execution with partial evaluation steps.

In this section we present how to extend the existing framework in a natural way to extract a specialized version for the verified program.

In Sect. 5.1 we introduce a bisimulation modality which allows us to relate two programs that behave indistinguishably on a given set of locations. The programs being related to each other are the original program and its specialized version. Sect. 5.2 defines the calculus rules for the newly added modal operator.

5.1 The Bisimulation Modality

Please note that several of the definitions given in this section assume that in program specialization the source language is the same as the target language. The definitions are generalizable to specialization (and finally compilation) between different languages (see Sect. 8).

Definition 5 (Location Sets). A location set is a set of

- program variables x or
- attribute expressions $o.a$ with $a \in \text{Attr}$ and o being a term of appropriate sort.

We are often not interested whether two states are identical, but rather whether they coincide on a given set of locations:

Definition 6. Let s_1, s_2 denote two states and loc a location set. We write $s_1 \sim_{loc} s_2$ if and only if for all $l \in loc$ it holds that $val_{D, s_1, \beta}(l) = val_{D, s_2, \beta}(l)$ where D denotes a first-order structure and β a variable assignment.

Specialized programs behave indistinguishably from the original program for (externally) observable locations. The set of observable locations includes usually all output variables and the part of the heap reachable from input and output variables. The formal definition bears a close relationship to the definition of non-interference:

Definition 7 (Observable Locations). *Let D denote a first-order structure and β a variable assignment. A location loc is called observable by a program \mathbf{p} if there are two states s_0, s_1 differing only in the evaluation of loc and either*

- *program \mathbf{p} terminates for s_i , but not for s_{1-i} ($i \in \{0, 1\}$), or*
- *program \mathbf{p} terminates for both states in final states $\{s'_i\} = \text{val}_{D, s_i, \beta}(\mathbf{p})$ ($i \in \{0, 1\}$) and there is a location loc' (not necessarily the same as loc) with $\text{val}_{D, s_i, \beta}(loc') \neq \text{val}_{D, s'_i, \beta}(loc')$ and $\text{val}_{D, s_i, \beta}(loc') \neq \text{val}_{D, s'_{1-i}, \beta}(loc')$*

A location loc is observable by a formula ϕ if there are two states s_1, s_2 differing only in loc with $\text{val}_{D, s_1, \beta}(\phi) \neq \text{val}_{D, s_2, \beta}(\phi)$.

Definition 8 (Bisimulates Relationship). *Let obs be a location set and s_1, s_2 two states with $s_1 \sim_{obs} s_2$. Two programs \mathbf{p}, \mathbf{q} are in a obs -bisimulation relation with respect to s_1 and s_2 if and only if for all first-order structures D and variable assignments β*

$$\text{val}_{D, s_1, \beta}(\mathbf{p}) \sim_{obs} \text{val}_{D, s_2, \beta}(\mathbf{q})$$

holds. We write $s_1, s_2 \models \mathbf{p} \sim_{obs} \mathbf{q}$.

If for all states s_1, s_2 with $s_1 \sim_{obs} s_2$ the statement $s_1, s_2 \models \mathbf{p} \sim_{obs} \mathbf{q}$ holds we simply write $\mathbf{p} \sim_{obs} \mathbf{q}$ and say \mathbf{p} obs -bisimulates \mathbf{q}

In this paper we restrict ourselves to program specialization and can use the same state s for s_1, s_2 ($s \sim_{obs} s$ holds trivially). The more general definition above is necessary when extending our approach to compilation (see Sect. [8](#)).

Lemma 1. *Let obs be the set of all locations observable by formula ϕ and let \mathbf{p}, \mathbf{q} be programs. If $s \models \mathbf{p} \sim_{obs} \mathbf{q}$ then for all first-order structures D and variable assignments β $D, s, \beta \models [\mathbf{p}]\phi \leftrightarrow [\mathbf{q}]\phi$ holds.*

Definition 9 (Bisimulation Modality—Syntax). *The bisimulation modality $[p_{src} \sim p_{target}]@(obs, use)$ is a modal operator providing compartments for the source program p_{src} , the target (or specialized) program p_{target} , and two location sets obs and use .*

We extend our definition of formulas: Let ϕ be a PL-DL formula and \mathbf{p}, \mathbf{q} two programs and obs, use two location sets, then $[\mathbf{p} \sim \mathbf{q}]@(obs, use)\phi$ is also a PL-DL formula.

Remark 1. The intended meaning of the location set use is to keep track of use-definition chains and contains roughly all locations that are read by program \mathbf{p} before they are redefined. The intent of set obs is to capture the locations observable by p and ϕ .

Remark 2. The definition above is tailored to the presentation of this paper, but in its general setting the modality can accommodate locations sets that represent arbitrary (local) analysis information.

We formalize our intuition by defining the semantics of the bisimulation modality:

Definition 10 (Bisimulation Modality—Semantics). *Let D, s, β denote a first-order structure, state and variable assignment, respectively. Further, \mathbf{p}, \mathbf{q} are programs and obs and use location sets.*

$val_{D,s,\beta}([\mathbf{p} \sim \mathbf{q}]@(obs, use)\phi) = tt$ if and only if

- (i) $val_{D,s,\beta}([\mathbf{p}]\phi) = tt$
- (ii) $s \models \mathbf{p} \sim_{obs} \mathbf{q}$
- (iii) obs is a superset of all locations observable by \mathbf{p} and ϕ
- (iv) $usedVar(s, \mathbf{p}, \phi) \subseteq use$ where $usedVar$ returns the set of variables read by \mathbf{p} or observed by ϕ before any redefinition (when executing \mathbf{p} in state s).

5.2 Sequent Calculus Rules for the Bisimulation Modality

The general sequent calculus rules for the bisimulation modality are of the following form:

ruleName

$$\Gamma_1 \Rightarrow \{\mathcal{U}_1\}[p_1 \sim q_1]@(obs_1, use_1)\phi_1, \Delta_1$$

...

$$\Gamma_n \Rightarrow \{\mathcal{U}_n\}[p_n \sim q_n]@(obs_n, use_n)\phi_n, \Delta_n$$

$$\frac{}{\Gamma \Rightarrow \{\mathcal{U}\}[p \sim q]@(obs, use)\phi, \Delta}$$

Unlike the normal sequent calculus rules which are executed in a bottom-up manner, the application of sequent calculus rules for the bisimulation modality consists of two phases.

1. Symbolic execution of source program p . It is performed bottom-up as usual in sequent calculus rules. In addition, the observable location sets obs_i are also propagated since they contain the locations observable by \mathbf{p}_i and ϕ_i that will be used in the second phase to synthesize the specialized program. Normally obs could contain the return variables of a method and the locations used in the continuation of the program.
2. We synthesize the target program \mathbf{q}_i and use_i by applying the rules in a top-down manner.

Based on the application of sequent calculus rules for the bisimulation modality, the process of synthesizing specialized programs is a two-phase procedure. The first phase is symbolic execution of the source program while keeping track of the observable location set obs . In the second phase, when the program is fully symbolically executed, the specialized program is synthesized by applying the rules in the other direction, starting with the `emptyBox` rule.

`emptyBox`

$$\frac{\Gamma \Rightarrow \{\mathcal{U}\}@(obs, _)\phi, \Delta}{\Gamma \Rightarrow \{\mathcal{U}\}[\mathbf{nop} \sim \mathbf{nop}]@(obs, obs)\phi, \Delta}$$

where $_$ is an anonymous placeholder, and **nop** explicitly denotes that the program is empty (no operation). The interesting aspect of this rule is that the location set *use* tracking read access to variables is set to *obs*, ensuring that observable locations are accessible in the specialized program.

Here are some examples of sequent calculus rules for the bisimulation modality. For convenience, we use \overline{p} to denote the specialized version of p .

assignLocalVariable

$$\frac{\Gamma \Rightarrow \{\mathcal{U}\}[1 := \mathbf{r}][\omega \sim \overline{\omega}]@(obs, use)\phi, \Delta}{\left(\begin{array}{ll} \Gamma \Rightarrow \{\mathcal{U}\}[l = r; \omega \sim l = r; \overline{\omega}]@(obs, use - \{l\} \cup \{r\})\phi, \Delta & \text{if } l \in use \\ \Gamma \Rightarrow \{\mathcal{U}\}[l = r; \omega \sim \overline{\omega}]@(obs, use)\phi, \Delta & \text{otherwise} \end{array} \right)}$$

The *use* set contains all program variables on which a read access might occur in the remaining program before being overwritten. In the first case, when the left side l of the assignment is among those variables, we have to update the *use* set by removing the newly assigned program variable l and adding the variable r which is read by the assignment. The second case makes use of the knowledge that the value of l is not accessed in the remaining program and skips the specialization of the assignment.

conditionalSplit

$$\frac{\begin{array}{l} \Gamma, \{\mathcal{U}\}b \Rightarrow \{\mathcal{U}\}[p; \omega \sim \overline{p; \omega}]@(obs, use_{p; \omega})\phi, \Delta \\ \Gamma, \{\mathcal{U}\}\neg b \Rightarrow \{\mathcal{U}\}[q; \omega \sim \overline{q; \omega}]@(obs, use_{q; \omega})\phi, \Delta \end{array}}{\Gamma \Rightarrow \{\mathcal{U}\}[\text{if } (b) \{p\} \text{ else } \{q\}; \omega \sim \text{if } (b) \{\overline{p; \omega}\} \text{ else } \{\overline{q; \omega}\}]@(obs, use_{p; \omega} \cup use_{q; \omega} \cup \{b\})\phi, \Delta}$$

(with b boolean variable.)

On encountering a conditional statement, symbolic execution splits into two branches, namely the **then**-branch and **else**-branch. The specialization of the conditional statement will result in a conditional. The guard is the same as used in the source program, **then**-branch is the specialization of the source **then**-branch continued with the rest of the program after the conditional, and the **else**-branch is analogous to the **then**-branch.

Note that the statements following the conditional statement are symbolically executed on both branches. This leads to duplicated code in the specialized program, and, potentially to code size duplication at each occurrence of a conditional statement. One note in advance: code duplication can be avoided when applying a similar technique as presented later in connection with the loop translation rule. However, it is noteworthy that the application of this rule might have also advantages: as discussed in [5], symbolic execution and partial evaluation can be interleaved resulting in (considerably) smaller execution trace. Interleaving symbolic execution and partial evaluation is orthogonal to the approach presented here and can be combined easily. In several cases this can lead to different and drastically specialized and therefore smaller versions of the remainder program ω and its specialization $\overline{\omega}$. The *use* set is extended canonically by joining the *use* sets of the different branches and the guard variable.

loopUnwind

$$\frac{\Gamma \Rightarrow \{\mathcal{U}\}[\text{if } (b) \{ \bar{p}; \text{while } (b) \{ p \} \} \omega \sim \text{if } (b) \{ \bar{p}; \text{while } (b) \{ p \} \} \omega] @ (obs, use) \phi, \Delta}{\Gamma \Rightarrow \{\mathcal{U}\}[\text{while}(b) \{ p \} \omega \sim \text{if } (b) \{ \bar{p}; \text{while}(b) \{ p \} \} \omega] @ (obs, use) \phi, \Delta}$$

whileInv

$$\frac{\Gamma \Rightarrow \{\mathcal{U}\} inv, \Delta \quad \Gamma, \{\mathcal{U}\} \{ \mathcal{V}_{mod} \} (b = \text{TRUE} \wedge inv) \Rightarrow \{\mathcal{U}\} \{ \mathcal{V}_{mod} \} [p \sim \bar{p}] @ (obs \cup use_1 \cup \{ b \}, use_2) inv, \Delta}{\Gamma, \{\mathcal{U}\} \{ \mathcal{V}_{mod} \} (b = \text{FALSE} \wedge inv) \Rightarrow \{\mathcal{U}\} \{ \mathcal{V}_{mod} \} [\omega \sim \bar{\omega}] @ (obs, use_1) \phi, \Delta} \quad \Gamma \Rightarrow \{\mathcal{U}\}[\text{while}(b) \{ p \} \omega \sim \text{while}(b) \{ \bar{p} \} \bar{\omega}] @ (obs, use_1 \cup use_2 \cup \{ b \}) \phi, \Delta$$

On the logical side the loop invariant rule is as expected and has three premises. Here we are interested in compilation of the analyzed program rather than proving its correctness. Therefore, it is sufficient to use *true* as a trivial invariant or to use any automatically obtainable invariant. In this case the first premise ensuring that the loop invariant is initially valid contributes nothing to the program compilation process and is ignored from here onwards (if *true* is used as invariant then it holds trivially).

Two things are of importance: the third premise executes only the program following the loop. Furthermore, this code fragment is not executed by any of the other branches and, hence, we avoid unnecessary code duplication. The second observation is that variables read by the program in the third premise may be assigned in the loop body, but not read in the loop body. Obviously, we have to prevent that the assignment rule discards those assignments when compiling the loop body. Therefore, we must add to the variable set *obs* of the second premise the used variables of the third premise and, for similar reasons, the program variable(s) read by the loop guard. In practice this is achieved by first executing the *use case* premise of the loop invariant rule and then using the resulting *use₁* set in the second premise. The work flow of the synthesizing loop is shown in Figure 4.

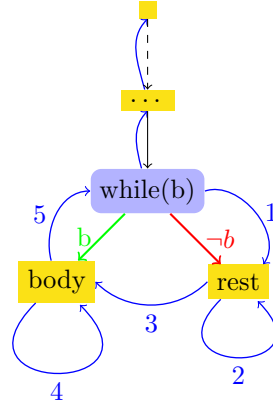


Fig. 4. Work flow of synthesizing loop

methodContract

$$\frac{\Gamma \Rightarrow \{\mathcal{U}\} \{ param_1 := v1 \parallel \dots \parallel param_n := vn \} pre, \Delta \quad \Gamma \Rightarrow \{\mathcal{U}\} \{ param_1 := v1 \parallel \dots \parallel param_n := vn \} \{ \mathcal{V}_{mod} \} (post \rightarrow [r = res; \omega \sim \bar{r} = \bar{res}; \bar{\omega}] @ (obs, use) \phi), \Delta}{\Gamma \Rightarrow \{\mathcal{U}\} [r = m(v1, \dots, vn); \omega \sim \bar{r} = \bar{res}; \bar{\omega}] @ (obs, use) \phi, \Delta}$$

Theorem 2 (Soundness Sequent Calculus Rules). *The rules for the bisimulation modality are sound.*

A proof sketch of the theorem is given in the appendix.

Theorem 3 (Soundness Procedure). *The procedure of program specialization by application of the sequent calculus for the bisimulation modality is sound.*

6 Application

In this section, we show the application of our framework to generate specialized programs for a Java-like language. Consider the program in Fig. 1. Our purpose is to specialize the `sum()` method which consists of non-trivial constructs such as attributes, a conditional, loop and method call. To achieve a clearer presentation we omit the postcondition ϕ following the bisimulation modality throughout the example as well as other unnecessary formulas in the sequents.

The first phase of our approach starts symbolically executing method `sum()` with the return value `tot` as the only observable location, i.e., $obs = \{\text{tot}\}$. The first statements of the method declare and initialize variables. These statements are executed similar to assignments. Altogether the `assignLocalVariable` rule is applied three times, where each assignment rule application is immediately followed by a partial evaluation step. We end up with

$$\frac{\frac{\frac{\Rightarrow \{\dots \parallel \text{tot} := 0\} [\text{while}(i \leq n) \dots \sim sp_3] @(\{\text{tot}\}, use_3)}{\Rightarrow \{\dots \parallel \text{count} := n\} [\text{tot} = 0; \text{while}(i \leq n) \dots \sim sp_2] @(\{\text{tot}\}, use_2)}}{\Rightarrow \{i := 0\} [\text{count} = n; \dots \sim sp_1] @(\{\text{tot}\}, use_1)}}{\Rightarrow [i = 0; \dots \sim sp_0] @(\{\text{tot}\}, use_0)}$$

where sp_i denotes the corresponding specialized program.

The next statement to be symbolically executed is the while loop computing the total sum. Instead of immediately applying the loop invariant rule, we unwind the loop once using the `loopUnwind` rule. Partial evaluation allows to simplify the guard $i \leq n$ and $i \geq 2 \ \&\& \ \text{cpn}$ of the introduced conditional to $i \leq 2$ and $0 \geq 2 \ \&\& \ \text{cpn}$ by applying constant propagation. Furthermore, the `then`-branch is eliminated because the guard $0 \geq 2 \ \&\& \ \text{cpn}$ can be evaluated to `false`. The result is as follows:

$$\frac{\frac{\frac{\Rightarrow \{i := 0 \parallel \dots \parallel \text{tot} := 0\} [\text{if}(0 \leq n) \{ \text{int } m1 = \text{read}(); \text{tot} = m1; i = 1; \text{while} \dots \} \sim sp_3] @(\{\text{tot}\}, use_3)}{\Rightarrow \{i := 0 \parallel \dots; \text{tot} := 0\} [\text{if}(0 \leq n) \{ \dots \text{tot} = 0 + m1; i = 1; \text{while} \dots \} \sim sp_3] @(\{\text{tot}\}, use_3)}}{\Rightarrow \{i := 0 \parallel \dots\} [\text{if}(0 \leq n) \{ \dots \text{if}(0 \geq 2 \ \&\& \ \text{cpn}) \dots; i = 0 + 1; \text{while} \dots \} \sim sp_3] @(\{\text{tot}\}, use_3)}}{\Rightarrow \{i := 0 \parallel \dots\} [\text{if}(i \leq n) \{ \dots \text{if}(i \geq 2 \ \&\& \ \text{cpn}) \dots; i ++; \text{while} \dots \} \sim sp_3] @(\{\text{tot}\}, use_3)}}{\Rightarrow \{i := 0 \parallel \dots \parallel \text{tot} := 0\} [\text{while}(i \leq n) \dots \sim sp_3] @(\{\text{tot}\}, use_3)}$$

Application of the `conditionalSplit` rule creates two branches. The `else`-branch contains no program so it is synthesized right away by applying the `emptyBox` rule. Symbolic execution of the `then`-branch, applies the `assignLocalVariable` rule three times until we reach the while loop again. We decide to unwind the loop a second time. The symbolic execution follows then the same pattern as before until we reach the loop for a third time. Fig. 5(a) shows the relevant part of the proof tree of the second loop unwinding.

Instead of unwinding the loop once more, we apply the loop invariant rule `whileInv`. The rule creates three new goals. The goal for the `init` premise is not of importance for the specialization itself, hence, we ignore it in the following.

The *used variables* set *use* of the `preserves` premise depends on the instantiation of the *use* set in the `use case` premise. To resolve the dependency we continue with the latter. In this case, the `use case` premise contains no program, so it is trivially synthesized by applying the `emptyBox` rule which results in `nop` as the specialized program and the only element `tot` in *obs* becomes the *use* set. Based on this, the *use* set of the `preserves` premise is the union of *obs*, `{tot}` and the locations used in the loop guard: `{tot, i}`. The program in the `preserves` premise is then symbolically executed by applying suitable rules until it is empty. This process is similar to that when executing the program in the `then`-branch of the conditional generated by `loopUnwind`. The proof tree resulting from the application of the loop invariant rule is shown in Fig. 5(b).

After symbolic execution we enter the second phase of our approach in which the specialized program is synthesized. Recall that when applying the `whileInv` rule, the procedure of synthesizing the loop starts with the `use case` branch. In our example, we have already performed this step and could already determine the instantiation of the observable location set *obs* of the `preserves` premise.

We explain now how the loop body is synthesized using the `preserves` premise: applying the `emptyBox` rule instantiates the placeholders sp_{12} and use_{12} with `nop` and `{tot, i}`. Going backwards, the `assignLocalVariable` rule tells us how to derive the instantiations for $sp_{11} = i++$; and $use_{11} = \{tot, i\}$. The instantiations for sp_{10} and use_{10} can be derived as `tot=tot+m; i++`; and `{tot, i}`. Before we can continue, the instantiations of sp_9 and use_9 need to be determined. Similar to the derivation of sp_{10} and use_{10} , applying the `assignLocalVariable` rule two times, we get $sp_9 = tot=tot+m*9/10; i++$; and $use_9 = \{tot, i\}$.

We have now reached the node where we previously applied the `conditionalSplit` rule. This rule allows us to derive `if(cpn) {tot=tot+m*0.9; i++;} else {tot=tot+m; i++;}`, as instantiation for sp_8 and `{tot, i, cpn}` as instantiation for use_8 . Applying suitable rules, we end up with the specialized program sp_6

```
while (i<=n) {
  int m = read();
  if (cpn) {tot=tot+m*9/10; i++;}
  else {tot=tot+m; i++; }
}
```

and the used variable set $use_6 = \{tot, i, cpn\}$.

$$\begin{array}{l}
 \frac{1 \leq n \Rightarrow \{\dots \| i := 2\} [\text{while}(i \leq n) \dots \sim sp_6] @(\{\text{tot}\}, use_6)}{\dots} \\
 \frac{0 \leq n \Rightarrow \{\dots\} [\text{if}(i \leq n) \dots; \text{while} \dots \sim sp_5] @(\{\text{tot}\}, use_5)}{\dots} \\
 \frac{0 \leq n \Rightarrow \{\dots \| m2 := \text{read}() \| \text{tot} := m2 \| i := 1\} [\text{while}(i \leq n) \dots \sim sp_5] @(\{\text{tot}\}, use_5)}{\dots} \\
 \dots \\
 \frac{\neg(0 \leq n) \Rightarrow \{\dots\} [\sim \text{nop}] @(\{\text{tot}\}, \{\text{tot}\}) \quad 0 \leq n \Rightarrow \{\dots\} [\text{int } m2 = \text{read}(); \dots \sim sp_4] @(\{\text{tot}\}, use_4)}{\Rightarrow \{\dots\} [\text{if}(0 \leq n) \{\text{int } m2 = \text{read}(); \text{tot} = m2; i = 1; \text{while} \dots\} \sim sp_3] @(\{\text{tot}\}, use_3)} \\
 \end{array}$$

(a) Specialization of the `while` loop via unwinding

$$\begin{array}{l}
 \dots \Rightarrow \{\dots \| i := i + 1\} [\sim sp_{12}] @(\{\text{tot}\} \cup \{i\}, use_{12}) \\
 \dots \Rightarrow \{\dots \| \text{tot} := \text{tot} + m\} [i ++; \sim sp_{11}] @(\{\text{tot}\} \cup \{i\}, use_{11}) \\
 \dots, \text{cpn} \Rightarrow \dots [\dots \sim sp_9] @(\{\text{tot}\} \cup \{i\}, use_9) \dots, \neg \text{cpn} \Rightarrow \{\dots\} [\text{tot} = \text{tot} + m; \dots \sim sp_{10}] @(\{\text{tot}\} \cup \{i\}, use_{10}) \\
 \dots \Rightarrow \{m := \text{read}()\} [\text{if}(\text{cpn}) \dots \sim sp_8] @(\{\text{tot}\} \cup \{i\}, use_8) \\
 \dots, \neg(i \leq n) \Rightarrow [\sim \text{nop}] @(\{\text{tot}\}, \{\text{tot}\}) \quad \dots, i \leq n \Rightarrow [\text{int} \dots \sim sp_7] @(\{\text{tot}\} \cup \{i\} \cup \{\text{tot}\}, use_7) \\
 \frac{1 \leq n \Rightarrow \{\dots \| i := 2\} [\text{while}(i \leq n) \dots \sim \{\text{tot}\}] @(\text{use}_6)}{\dots} \\
 \end{array}$$

(b) Specialization of the `while`-loop using the loop invariant rule

 Fig. 5. Specialization of the `while`-loop by different means

Following the symbolic execution tree backwards and applying the corresponding rules, we finally synthesize the specialized program for `sum()` as follows:

```
public int sum(int n) {
  int i; int tot; tot = 0;
  if (0 <= n) {
    int m1 = read(); tot = m1;
    if (1 <= n) {
      int m2 = read();
      tot = tot + m2; i = 2;
      while(i <= n) {
        int m = read();
        if (cpn) { tot = tot + m * 9 / 10; i++; }
        else { tot = tot + m; i++; }
      } }
    return tot; }
}
```

7 Related Work

JSpec [15] is a program specializer for Java and, therefore, has the same goal as our approach. In fact, JSpec is not working with full Java but a subset without concurrency, dynamic loading, etc. In this sense it is similar to our work. However, they use an *offline* partial evaluation technique that depends on *binding time analysis*. Our work is based on symbolic execution to derive information on-the-fly, similar to *online* partial evaluation [14]. Our work is related to the latter, the main difference being that we do not generate the specialized program during the symbolic execution phase, but synthesize it in the second phase. In principle, our first phase can obtain as much information as *online* partial evaluation, and the second phase can generate a more precise specialized program.

Our approach is also related to the *Verifying Compiler* [11] project which aims at the development of a compiler that verifies the program during compilation. In contrast to this, our approach might be called instead the *Compiling Verifier*. Like our work, compiler verification [8] aims to guarantee the correctness of the target program. The difference is that compiler verification attempts to verify the compiling program which is very expensive and hardly scales to realistic target languages and sophisticated optimizations.

Our work is closely related to rule-based compilation [14]. It differs in the sense that to the best of our knowledge their inference machine is by far not as powerful as the mature simplification engine used in KeY. Also closely related are recent approaches to translation validation of optimizing compilers (e.g., [2]) which also use a theorem prover to discharge proof obligations. They work usually on an abstraction of the target program. Both mentioned approaches encode the compilation strategy within the rules, while our approach separates the actual strategy from the translation rules. What distinguishes our work from

most approaches that we know is that the starting point is a system for functional verification of Java which is used for program specialization in such a way that it becomes fully automatic.

8 Conclusion and Future Work

We presented a novel approach to specialize programs via a software verification tool in a two-phase manner. In the first phase, symbolic execution interleaved with simple partial evaluation is performed. Symbolic execution permits dynamic analysis at compile time which is similar to online partial evaluation. In the second phase, the specialized program is synthesized. A use-definition chain set is maintained to eliminate unused assignments and to avoid unnecessary statements occurring in the specialized program. The correctness of the specialization is guaranteed by the bisimulation relationship of the source and specialized programs, together with the soundness of the program logic. It is a new architecture to construct verified compilers by combining verification, partial evaluation and local transformation. The implementation is currently ongoing with KeY tool and more results will be reported later.

Although this approach is defined for a Java-like language, it will be interesting to see whether other features such as concurrency could be handled, going towards full-Java.

Orthogonally, there are still opportunities to optimize the procedure. For instance, on encounter of a loop, the heuristics that decide whether to unwind it or not have a strong influence on the resulting specialized programs. Importing information, e.g., loop invariants, from other tools could also be useful.

The idea of this paper is to generate specialized programs, however, the bisimulation modality is not restricted to source and target program being from the same language, but it can be generalized to other languages provided with corresponding observable locations. Consequentially, the approach is still sound for generating bytecode or other intermediate languages. We plan to apply our approach to the modeling language ABS developed in the context of the HATS project [\[6,7\]](#)

Furthermore, the close connection between the program logic and compilation allows to ensure the correctness of the compilation process as such. We see a great potential of our approach when encoding security or safety properties in terms of pre-/postconditions. This should allow to identify unsafe or unsecured execution paths during compilation and either to abort compilation or to wrap the undesired execution paths in a wrapper that at least ensures the safety or security property of interest. For example, execution paths that may leak information can be secured by omitting the assignments that violate secure information flow. Another possibility would be to ensure that if the program enters an unsecured execution path, then the program will not terminate. Exploring these avenues is future work.

Acknowledgments. We thank Wolfgang Ahrendt for fruitful discussions as well as for valuable comments on an earlier version of this paper.

References

1. Augustsson, L.: A compiler for lazy ML. In: Proceedings of the 1984 ACM Symposium on LISP and functional programming, LFP 1984, pp. 218–227. ACM, New York (1984)
2. Barrett, C.W., Fang, Y., Goldberg, B., Hu, Y., Pnueli, A., Zuck, L.D.: TVOC: A translation validator for optimizing compilers. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 291–295. Springer, Heidelberg (2005)
3. Beckert, B., Hähnle, R., Schmitt, P. (eds.): Verification of Object-Oriented Software. The KeY Approach. LNCS (LNAI), vol. 4334. Springer, Heidelberg (2007)
4. Breebaart, L.: Rule-based compilation of data parallel programs. PhD thesis, Delft University of Technology (2003)
5. Bubel, R., Hähnle, R., Ji, R.: Interleaving symbolic execution and partial evaluation. In: de Boer, F.S., Bonsangue, M.M., Hallerstede, S., Leuschel, M. (eds.) FMCO 2009. LNCS, vol. 6286, pp. 125–146. Springer, Heidelberg (2010)
6. Clarke, D., Diakov, N., Hähnle, R., Johnsen, E.B., Schaefer, I., Schäfer, J., Schlatte, R., Wong, P.Y.H.: Modeling spatial and temporal variability with the HATS abstract behavioral modeling language. In: Bernardo, M., Issarny, V. (eds.) SFM 2011. LNCS, vol. 6659, pp. 417–457. Springer, Heidelberg (2011)
7. Clarke, D., Muschewici, R., Proença, J., Schaefer, I., Schlatte, R.: Variability modelling in the ABS language. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) FMCO 2010. LNCS, vol. 6957, pp. 206–226. Springer, Heidelberg (2011)
8. Dave, M.A.: Compiler verification: a bibliography. SIGSOFT Softw. Eng. Notes 28, 2 (2003)
9. Harel, D., Kozen, D., Tiuryn, J.: Dynamic Logic. MIT Press, Cambridge (2000)
10. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM 12(10) (October 1969)
11. Hoare, T.: The verifying compiler: A grand challenge for computing research. J. ACM 50, 63–69 (2003)
12. Jones, N., Gomard, C., Sestoft, P.: Partial Evaluation and Automatic Program Generation. Prentice Hall, New York (1993)
13. King, J.C.: A program verifier. PhD thesis, Carnegie-Mellon University (1969)
14. Ruf, E.S.: Topics in online partial evaluation. PhD thesis, Stanford University, Stanford, CA, USA, UMI Order No. GAX93-26550 (1993)
15. Schultz, U.P., Lawall, J.L., Consel, C.: Automatic program specialization for Java. ACM Trans. Program. Lang. Syst. 25(4), 452–499 (2003)

Appendix: Proof of Lemma 2

Proof (Sketch). We give here only the proof for `conditionalSplit`. The proofs for other rules are similar. To prove soundness of a rule we need to show that the validity of the conclusion is a consequence of the premises’ validity.

Let D, s_a, β be arbitrary, but fixed. We assume that $val_{D, s_a, \beta}(\Gamma \wedge \neg \Delta) = tt$ otherwise we are trivially done. We have now to prove that

$$val_{D, s_a, \beta}(\{U\} [\text{if } (b) \{p\} \text{ else } \{q\}; \omega \sim \text{if } (b) \{\overline{p}; \overline{\omega}\} \text{ else } \{\overline{q}; \overline{\omega}\}] @ (obs, use_{p; \omega} \cup use_{q; \omega} \cup \{b\}) \phi)$$

holds or, equivalently, that

$$val_{D,s,\beta}([\text{if } (b) \{p\} \text{ else } \{q\}; \omega \sim \text{if } (b) \{\overline{p}; \overline{\omega}\} \text{ else } \{\overline{q}; \overline{\omega}\}] @ (obs, use_{p;\omega} \cup use_{q;\omega} \cup \{b\}) \phi)$$

with $val_{D,s_a,\beta}(U) = s$ holds.

We have to check that the four requirements stated in Def. [10](#) are satisfied. First, we need to check requirement [\(i\)](#), namely:

$$val_{D,s,\beta}(\{\mathcal{U}\}[\text{if } (b) \{p\} \text{ else } \{q\}; \omega]) = tt$$

This requirement is equivalent to the soundness proof of the conditional rule for the standard calculus version and skipped.

The most interesting requirement to be checked is [\(ii\)](#). We need to show that the specialized program *s-obs-bisimulates* the original program:

$$s \models \text{if } (b) \{p\} \text{ else } \{q\}; \omega \sim_{obs} \text{if } (b) \{\overline{p}; \overline{\omega}\} \text{ else } \{\overline{q}; \overline{\omega}\}$$

Case $val_{D,s,\beta}(b) = true$: Validity of the first premise ensures that

$$s \models p; \omega \sim_{obs} \overline{p}; \overline{\omega}$$

which means according to its definition

$$val_{D,s,\beta}(p; \omega) \sim_{obs} val_{D,s,\beta}(\overline{p}; \overline{\omega})$$

With that we get

$$\begin{aligned} val_{D,s,\beta}(\text{if } (b) \{p\} \text{ else } \{q\}; \omega) &= val_{D,s,\beta}(p; \omega) \\ &\sim_{obs} val_{D,s,\beta}(\overline{p}; \overline{\omega}) \\ &= val_{D,s,\beta}(\text{if } (b) \{\overline{p}; \overline{\omega}\} \text{ else } \{\overline{q}; \overline{\omega}\}) \end{aligned}$$

The second case $val_{D,s,\beta}(b) = false$ is analogous. Taking both cases we can conclude the proof of requirement [\(ii\)](#).

Requirement [\(iii\)](#) is satisfied if

$$use_{p;\omega} \cup use_{q;\omega} \cup \{b\}$$

is a superset of all observable locations of $\text{if } (b) \{p\} \text{ else } \{q\}; \omega$ and ϕ . From the premises we get directly that $use_{p;\omega}$ and $use_{q;\omega}$ are supersets of all observable locations of the branches, the remaining program and formula ϕ . The only additional location which is read by the conditional statement except those of its branches and which may not yet be included is variable b . The union of all these sets is the set used in the rule's conclusion and satisfies obviously requirement [\(iii\)](#). Finally, we need to check requirement [\(iv\)](#) which can be done analogous to the check for requirement [\(iii\)](#). \square

Model-Based Analysis Tools for Component Synthesis

Luigia Petre, Kaisa Sere, and Leonidas Tsiopoulos

Department of Information Technologies, Åbo Akademi University, Turku, Finland

Abstract. Component-based development typically refers to assembling pre-existing pieces of software or hardware for integrating them into new systems. In this paper we introduce a formalism-based approach to verify the component boundaries, based on the component interdependencies. We base this synthesis method on B Action Systems and the animation techniques provided by the ProB tool. In addition, we put forward another applicability for our method, namely to mapping components to hardware platform tiles.

Keywords: Component Synthesis, B Action Systems, ProB tool, Application Mapping.

1 Introduction

Component-based development identifies and manages interdependencies among preexisting software or hardware parts for integrating them into new systems [1], [2]. During the last two decades, it has been widely recognized that methodologies and frameworks with adequate tool support are needed, in order to facilitate the component-based development.

One of the candidate approaches for specifying *reliable* software and hardware systems, modelling the communication of their components, as well as verifying their design is provided by *formal methods* [3], [4]. These methods are often accompanied by adequate tool support making them more accessible. The importance of formal methods for the development of complex systems is justified by the potential they have in avoiding costly errors in the later design phases, hence, contributing to the *reliability* of such systems. A formal component-based development framework additionally to specification and verification of complex systems needs to provide means for component interdependency analysis and component synthesis with adequate tool support.

In this paper, we propose a formal method based methodology for component synthesis. We understand the functionality of a component as a collection of *services* that the component has to implement [2]. Components have been traditionally developed with an emphasis on the specification of their functionality, without explicitly describing and analysing their interdependencies. Here, based on the services that components need to implement, we identify some initial boundaries of components. Then, we identify those services needed for communication among components; hence, we identify the component interdependencies.

Based on these interdependencies, we then reason about the suitability of certain boundaries, i.e., we argue whether some components should or not merge. As an application of our approach, we employ these interdependencies to place components on hardware platforms, so that highly communicating components find each other in their vicinity.

We base our approach on two formalisms. First, we employ a state-based formal method, B Action Systems [5], created for reasoning about parallel and distributed systems within the B Method [6]. We start our approach based on a B Action Systems model comprising a predefined grouping of services into components. The purpose of our method is to determine configurations that are more suitable if possible. Second, we employ a model checker also based on the B Method principles, namely the ProB tool [7] that provides various verification techniques such as model checking and animation. We argue that animation of component services is a valuable technique for identifying the component interdependencies. Based on it, we devise a method for challenging the component boundaries. An animation of the model generates the computed coverage, i.e., the *guaranteed* as well as the *possible* relations between the services of the components. We analyze these relations and decide on new component configurations taking into account a desired number for component interdependencies. We continue applying the method on a component configuration until we find a suitable grouping of services into components with respect to the desired number for component interdependencies.

In addition, we employ the animating tool GeneSyst [8] to further assist our synthesis method to the placement of components on hardware platforms. This application of the method involves some adaptation as well as the incorporation of additional ProB animating facilities. The latter contribute to determining the execution traces and the interdependency details of the components. Instead of defining new components, we simply place highly communicating components as close as possible to each other for efficient communication.

We proceed as follows. In Section 2, we describe B Action Systems and the animating tools to the extent needed in this paper; we also describe our case study and model it formally. In Section 3, we introduce our proposed synthesis method, exemplify it, and discuss it in detail. We then adapt our proposed method to mapping application components to hardware platform tiles in Section 4. We discuss related work in Section 5 and we conclude in Section 6.

2 Preliminaries

In this section we present the formalisms we employ in this paper together with an example modeled with these formalisms.

2.1 B Method Based Formalisms

B Action Systems [5] is a state-based formalism based on Action Systems [9] and the B Method [6]. This framework was developed in order to allow reasoning about parallel and distributed systems within the B Method.

We illustrate the form of a B Action System in Fig. 1 (a). The main computational unit is the *machine*, identified by a unique name A. A machine has a finite set of (local) variables and a finite set of operations that evaluate and modify the variables. The variables of the machine are declared in the **VARIABLES**-clause; their values describe the *state* of the machine. The **INVARIANT**-clause defines the types of the variables and gives their guaranteed behaviour. Initial values are assigned to the variables in the **INITIALISATION**-clause. The operations in the **OPERATIONS**-clause are of the form `Oper = SELECT P THEN S END`, where P is a predicate (called *guard*) on the variables and S is a substitution (update) statement; hence, an operation can evaluate and modify the state of the system modelled by the variable values. When P holds, the operation Oper is said to be enabled. Only enabled operations are considered for execution and if there are several operations enabled simultaneously then only one is selected for execution in a non-deterministic manner. If some operations have no variables in common and are enabled at the same time, then they can be considered to execute in parallel since their sequential execution in any order gives the same result. When there are no enabled operations the machine terminates.

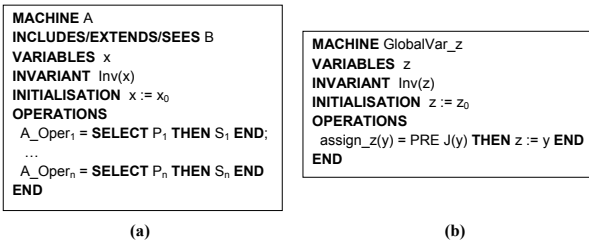


Fig. 1. (a) Example of a B Action System, (b) Global variables in B Action Systems

Structuring mechanisms can be used to express B action systems as a composition of subsidiary systems. The mechanisms used in this paper are the **SEES** and **INCLUDES** mechanisms [6]. The **SEES**-mechanism allows reading access to the seeing machine, meaning that variables of the seen machine can be used in the initialization and operations of the seeing machine. When using the **INCLUDES**-mechanism, in addition to the reading access, the invariant of the including machine can express requirements on the variables of the included machine. The variables of the included machine are directly visible to the including machine, but they may only be updated via the included system.

Global variables that can be read and updated by more than one system are one of the most common communication mechanisms in B Action Systems. A global variable z is declared in a separate machine as shown in Fig. 1 (b). This machine is then **INCLUDED** or **SEEN** in the machines that refer to the global variable and the including machine is allowed to assign a new value y to the variable z via an operation `assign_z(y)` [5] in the global variable machine. Global variables are further employed in the next section.

Parallel composition of several machines into a single machine is an important aspect of our modelling approach as will become apparent in the following

sections. The parallel composition of a machine A and a machine B is illustrated in Fig. 2 and it is formed by merging the variables, invariants and operations of A and B. The local variables of the machines have to be distinct. This can easily be achieved by renaming the conflicting variables before forming the composition. The global variables declared in a global variable machine will be the global variables of the parallel composition. Since the invariant of the composed machine is the conjunction of the invariants of the individual machines before the composition, the operations of each machine should preserve the invariants of all other machines in the composition. This is mainly a restriction on the assignments to the common global variables.

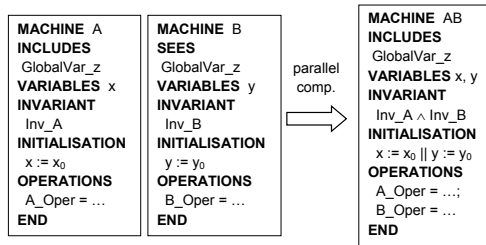


Fig. 2. Parallel composition of B Action Systems A and B

ProB Tool. In order to analyse B Action System specifications we employ the ProB tool [7]. ProB is a model checking and animation tool for B machines that includes a fully automatic animator written in SICStus prolog [10]. ProB takes an instantiated model in B, i.e. a model in which any generic set has been instantiated with some concrete values to avoid state explosion and generates a finite coverage graph. In this paper we only employ the animation capabilities provided by ProB.

ProB also computes the *coverage* of the model after a random animation of a number of operation executions. The computed coverage consists of information about the total number of the nodes of the state space of the system and their status (*deadlocked*, *live*, or *open*). It also provides the total number of operations of the model, as well as the possibly covered operations between the nodes. The list of **DEFINITELY_ENABLED_AFTER** operations and the list of **POSSIBLY_ENABLED** operations are also provided. The former list contains the operations that are definitely enabled after the execution of a subset of the model's operations. The latter list contains the operations that are possibly enabled after the execution of each operation of the model. Furthermore, the *Signature-Merge Reduced Statespace* (S-MRS) graph contains specific transitions (operation executions) based on some specific global state, for instance, we can observe the different starting points of possible interleaving. This S-MRS graph corresponds to the full state-space graph.

GeneSyst Tool. S-MRS graphs from ProB are prone to rapid expansion if the model is non-deterministic and the sets defined for variable declarations are

large. Such graphs can then be difficult to examine and of small value to the user of the tool. A potential alleviation of this problem comes from animating tools like GeneSys [8]. This is an animating tool for B machines that takes as input a set of disjoint predicates (given by the user in an additional **ASSERTIONS**-clause) on the state variables of a B machine. Based on several internal checks and proofs, the tool outputs an abstract state-transition diagram corresponding to the predicates given by the user.

2.2 An Example Modelled with the Formalisms

As a case study, we consider a model of an asynchronous pipelined processor introduced by Plosila and Sere [11] and specified originally within the Action Systems formalism [9]. In the following we describe this model to the extent needed in this paper and put forward the B Action Systems and ProB specifications of this model.

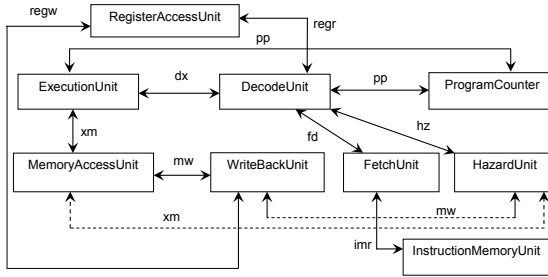


Fig. 3. Asynchronously communicating subsystems of a pipelined processor

In Fig. 3 we illustrate several components of the processor together with their dependencies: FetchUnit, InstructionMemoryUnit, DecodeUnit, RegisterAccessUnit, ProgramCounter, HazardUnit, ExecutionUnit, MemoryAccessUnit and WriteBackUnit. The components cooperate in the following manner. First, FetchUnit fetches a new instruction for manipulation from InstructionMemoryUnit and initiates the communication with DecodeUnit. DecodeUnit then communicates with HazardUnit in order to avoid a *pipeline hazard*, i.e., a read-write conflict on the relevant registers between a new incoming instruction in DecodeUnit and a current instruction being written to the memory. When the memory is updated with the current instruction, HazardUnit acknowledges back and allows DecodeUnit to increment ProgramCounter, read the register, and decode the new instruction. DecodeUnit waits for the acknowledgements from ProgramCounter and RegisterAccessUnit before it acknowledges the communication with FetchUnit and sends the current decoded instruction to ExecutionUnit. After the communication with FetchUnit is acknowledged, a new instruction can be fetched. ExecutionUnit manipulates the instruction: depending on its type, it increments or loads ProgramCounter. After the acknowledgement from ProgramCounter, ExecutionUnit initiates communication with MemoryAccessUnit and acknowledges the communication with DecodeUnit, so

that it can receive a new decoded instruction. MemoryAccessUnit sends the instruction to WriteBackUnit, that writes the instruction to the memory through Register-AccessUnit and acknowledges back to MemoryAccessUnit the memory update. The manipulation of the next instruction can then continue from an intermediate pipelined stage. More details of the model appear in [11].

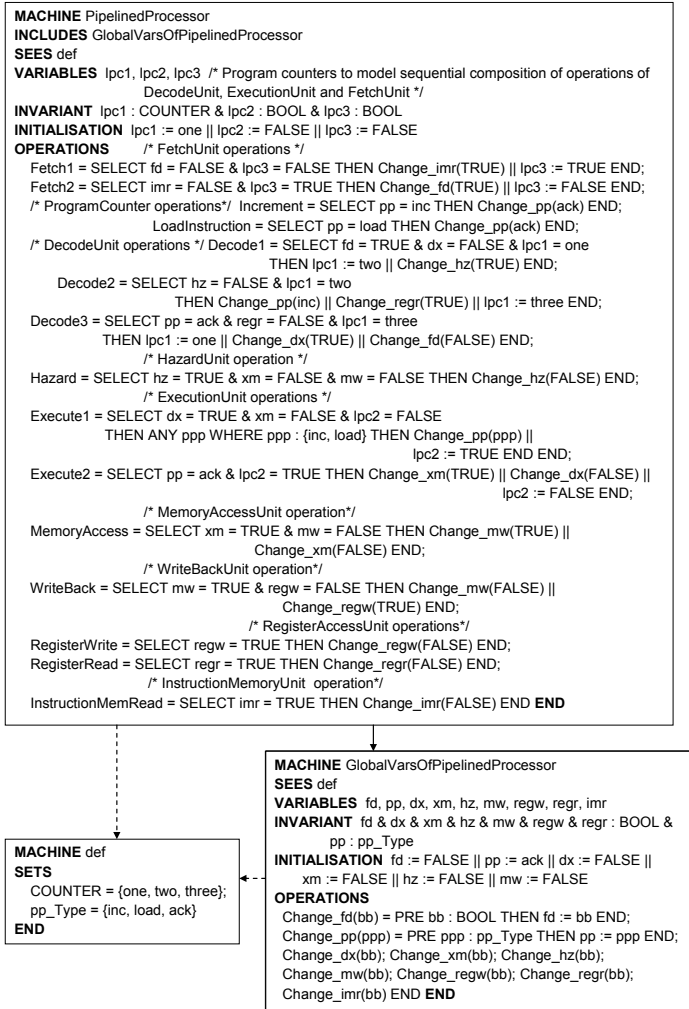


Fig. 4. B Action System of PipelinedProcessor

The dependencies between the components are modeled with global variables. The dashed arrows indicate read access to the attached variables *mw* and *xm* in HazardUnit. The continuous line arrows indicate read and write access to the attached variables. For example, the components ProgramCounter, DecodeUnit and

ExecutionUnit are all allowed to update the global communication variable `pp`, that can be updated with the values `inc` (to increment `ProgramCounter`), `load` (to load `ProgramCounter`), and `ack` (acknowledgement from `ProgramCounter` for the two possible updates). The other global variables are of type `BOOL` to model the request and acknowledgement phases of the asynchronous communication. The value `TRUE` corresponds to the *request* phase and the value `FALSE` corresponds to the *acknowledgement* phase.

For applying the analysis techniques provided by ProB, the decomposed model of the processor illustrated in Fig. 3 is composed (in parallel) into a single B Action System as defined in Section 2.1. Currently, ProB can model check and animate only one system and not a collection of systems running in parallel at the same hierarchical level. However, this is not a real restriction since the execution of various systems in parallel is equivalent to their parallel composition as explained in Section 2.1.

In Fig. 4 we illustrate the specification of the pipelined processor. The shared global variables together with simple operations to update them are defined in the separate global variable `GlobalVarsOfPipelinedProcessor`. The required sets for this specification are defined in a separate machine `def`. The continuous line arrow indicates the **INCLUDES** structuring mechanism and the dashed line arrow indicates the **SEES** mechanism. `PipelinedProcessor` has three (local) variables named `lpc1`, `lpc2` and `lpc3` to model the sequential composition of the operations of `DecodeUnit`, `ExecutionUnit` and `FetchUnit`, respectively. The operations of each component are grouped together and the components are conventionally separated from each other with a comment line. For instance, the operations between the comment lines `/* FetchUnit operations */` and `/* ProgramCounter operations */` denote the `FetchUnit`, the operations between the comment lines `/* ProgramCounter operations */` and `/* DecodeUnit operations */` denote the `ProgramCounter` and so on. The functionality of the subsystems in Fig. 4 corresponds to the description of their dependencies in Fig. 3.

We observe here the nature of this case study. In the model illustrated in Fig. 3, the variables are stored within the various components and declared global, so that other components can also access them. The control flow in the pipelined processor is mainly sequential and then generates various branches of parallel flows. One component can start its job only if specific actions have taken place before that. We translate this component sequencing to communication modeled in B Action Systems via the **SEES** and **INCLUDES** mechanisms.

We now observe the nature of the B Action Systems model of `PipelinedProcessor`. The most obvious difference with respect to the model illustrated in Fig. 3 is that the global variables are no longer stored within the units but instead in one **INCLUDED** machine. This is only a technical detail for specifying `PipelinedProcessor` in ProB. When we analyse the pipelined application, we assume the variables stored within the components as in the original model [11] illustrated in Fig. 3. In Fig. 5, we show the history of the executed operations after an animation; this is only a part of the main interface of ProB, where the model in Fig. 4 is specified. We focus on this part of the window because it is the most relevant for



Fig. 5. Implementation of PipelinedProcessor in ProB and history of executed operations

our component mapping method. The other parts of the main interface contain the specification under animation as well as the enabled operations and the state properties during animations.

3 The Synthesis Method

Our approach to synthesising components is based on the component services: they are the basic units of composition. We model component services with machine operations in the B Action Systems model; hence, we start our approach based on a B Action Systems model comprising all the possible operations to synthesize. Every specification typically comes with a semantical grouping of operations into components. We take advantage of this initial grouping; the purpose of our method is to determine more suitable component configurations if possible. We assume we are given a positive integer n that represents the maximum desired number of interdependencies among components. Hence, the inputs of our method are a suggested delineation of components and the number n .

The B Action Systems model is translated into a ProB model on which we apply various animations techniques in order to challenge the component boundaries. Our proposed method is described below.

1. We run a (large enough) animation in ProB, based on the ProB model of the application. Based on the computed coverage of the ProB model, we record the relations between the operations in the lists `DEFINITELY_ENABLED_AFTER` and `POSSIBLY_ENABLED`.
2. We replace each operation in the recorded lists of Step 1 with the name of its component.
3. We count the number of occurrences of the name of each component in the recorder lists created in Step 2. We store this information in a table with three columns: one for the component names, one for the number of communications of each component, modeled by the enabledness relations recorded in Steps 1-2, and one for the names of the communication

partners of each component. If the same relation occurs in both lists DEFINITELY_ENABLED_AFTER and POSSIBLY_ENABLES, we only count it once. We place the component names in the table in the descending order of the number of their communications.

4. We check the table created in the previous step to identify components that communicate more than n times with other components, by observing the number of communications associated with component names in the table.
 - (a) If there is no such component, then we have a suitable component configuration and the method ends.
 - (b) If there are such components, then they have a too narrow border. We enlarge the border of the highest communicating component to comprise all the services of all its communication partners. We feed back the new component configuration to our method and restart from Step 1.

We run a large enough animation so that it produces a full state-space graph in ProB. By analysing applications in the above way we have a structured method for analysing the number of component interdependencies. Based on it we can change the component configuration to a desired level of component interdependencies.

3.1 Applying the Synthesis Method

As a case study of our method we study the component interdependencies of the pipelined application described in Section 2.2. We apply our synthesis method starting with the ProB specification, where the components are delineated by comments as shown in Fig. 4. We assume that $n = 4$. When applying Step 1 in our method we get the lists described in the left part of Fig. 6.

After applying Steps 2 and 3, we obtain the table shown in Fig. 7. This table shows for each component the number of communications with other components as well as the names of these components. The definite and possible dependencies between the components are shown in the right part of Fig. 6. Studying the table we can apply Step 4 of our method, where we reach the conclusion that `DecodeUnit` should be merged with `FetchUnit`, `RegisterAccessUnit`, `ProgramCounter`, `HazardUnit`, and `ExecutionUnit` in order to have more suitable component boundaries.

The method should now be applied again on the new component configuration. We do not show it here due to lack of space, but one can easily see that in the new configuration there will only be four components, with at most four communications each. Hence, the method now stops.

3.2 On the Correctness of Our Method

The method takes as input a given component configuration $\mathcal{C} = \{C_1, \dots, C_m\}$ so that each component C_i , $i \in \{1, \dots, m\}$ implements a certain number p_i of services S_j , $j \in \{1, \dots, p_i\} : C_i = \{S_j | j \in \{1, \dots, p_i\}\}$. Some of these services may denote internal computation, while others may enable some of the services of other components. The latter occurrence is detected by our method via the

<p>DEFINITELY_ENABLED_AFTER Hazard==>Decode2 Decode2==>Increment Decode2==>RegisterRead Decode3==>Fetch1 Decode3==>Execute1 Execute2==>MemoryAccess MemoryAccess==>WriteBack Fetch1==>InstructionMemRead WriteBack==>RegisterWrite InstructionMemRead==>Fetch2 LoadInstruction==>Execute2</p> <p>POSSIBLY_ENABLES Hazard?=>Decode2 Decode2?=>Increment Decode2?=>RegisterRead RegisterRead?=>Decode3 Decode3?=>Fetch1 Decode3?=>Execute1 Decode1?=>Hazard Execute2?=>MemoryAccess Execute2?=>Decode1 Increment?=>Execute2 Increment?=>Decode3 MemoryAccess?=>WriteBack Fetch1?=>InstructionMemRead WriteBack?=>RegisterWrite InstructionMemRead?=>Fetch2 RegisterWrite?=>Hazard Fetch2?=>Decode1 LoadInstruction?=>Execute2 Execute1?=>Increment Execute1?=>LoadInstruction</p>	<p>DEFINITELY_ENABLED_AFTER HazardUnit ==> DecodeUnit DecodeUnit ==> ProgramCounter DecodeUnit ==> RegisterAccessUnit DecodeUnit ==> FetchUnit DecodeUnit ==> ExecutionUnit ExecutionUnit ==> MemoryAccessUnit MemoryAccessUnit ==> WriteBackUnit FetchUnit ==> InstructionMemoryUnit WriteBackUnit ==> RegisterAccessUnit InstructionMemoryUnit ==> FetchUnit ProgramCounter ==> ExecutionUnit</p> <p>POSSIBLY_ENABLES HazardUnit ?=> DecodeUnit DecodeUnit ?=> ProgramCounter DecodeUnit ?=> RegisterAccessUnit RegisterAccessUnit ?=> DecodeUnit DecodeUnit ?=> FetchUnit DecodeUnit ?=> ExecutionUnit DecodeUnit ?=> HazardUnit ExecutionUnit ?=> MemoryAccessUnit ExecutionUnit ?=> DecodeUnit ProgramCounter ?=> ExecutionUnit ProgramCounter ?=> DecodeUnit MemoryAccessUnit ?=> WriteBackUnit FetchUnit ?=> InstructionMemoryUnit WriteBackUnit ?=> RegisterAccessUnit InstructionMemoryUnit ?=> FetchUnit RegisterAccessUnit ?=> HazardUnit FetchUnit ?=> DecodeUnit ProgramCounter ?=> ExecutionUnit ExecutionUnit ?=> ProgramCounter ExecutionUnit ?=> ProgramCounter</p>
---	--

Fig. 6. Part of the computed coverage of PipelinedProcessor

Components	No of Component Interactions	Communication Partners
DecodeUnit	10	FetchUnit, RegisterAccessUnit, ProgramCounter, ExecutionUnit, HazardUnit
ExecutionUnit	5	DecodeUnit, MemoryAccessUnit, ProgramCounter
FetchUnit	4	DecodeUnit, InstructionMemoryUnit,
RegisterAccessUnit	4	DecodeUnit, WriteBackUnit, HazardUnit
ProgramCounter	3	DecodeUnit, ExecutionUnit
HazardUnit	3	DecodeUnit, RegisterAccessUnit
MemoryAccessUnit	2	ExecutionUnit, WriteBackUnit
InstructionMemoryUnit	2	FetchUnit
WriteBackUnit	2	MemoryAccessUnit, RegisterAccessUnit

Fig. 7. Dependencies between the components of PipelinedProcessor, their number of interactions and their communication partners

ProB animation tools and we refer to it as *component communication*. We identify with our method a certain number r_i for each component $C_i, i \in \{1, \dots, m\}$ that denotes the number of definite and possible communications for the respective component $C_i, i \in \{1, \dots, m\}$. We denote with max_k the highest number of component communications at run k of the method, for instance $max_1 = 10$ and $max_2 = 4$ in our example above. We also take as input to the method a positive integer n , denoting the maximum desired number of component interdependencies. The goal of the method is to determine a component configuration $C' = \{C'_1, \dots, C_{m'}\}$ where each $r'_i \leq n, i \in \{1, \dots, m'\}$.

Termination. As we start with a finite number m of components and each rerun of the method is triggered by some of these components merging, we observe that $m < m'$. This necessarily means that the method terminates, in the extreme case with $m' = 1$ and $r'_i = 0$. The extreme case thus translates to all the components merged into one.

From One Configuration to the Next. While we are certain that the method terminates with $0 \leq r'_i \leq n, i \in \{1, \dots, m'\}$, we cannot guarantee that the highest number max_k of component communications at each run k of the method is continuously decreasing. For instance, Fig. 8 shows a configuration $\mathcal{C} = \{C_1, C_2, C_3, C_4, C_5, C_6\}$ where $r_1 = 20, r_2 = 18, r_3 = 18, r_4 = 16, r_5 = 12,$ and $r_6 = 12$ and $n = 12$. We also assume that C_1 communicates 10 times with C_2 , 6 times with C_3 , and 4 times with C_4 ; C_2 communicates 10 times with C_1 , 4 times with C_3 , and 4 times with C_5 ; C_3 communicates 6 times with C_1 , 4 times with C_2 , and 8 times with C_6 ; C_4 communicates 4 times with C_1 , 8 times with C_5 , and 4 times with C_6 ; C_5 communicates 4 times with C_2 and 8 times with C_4 ; and C_6 communicates 8 times with C_3 and 4 times with C_4 . This means that after the first run of the method, the first four components are merged; we have $max_1 = 20$ while $max_2 = 24$.

Components	No of Component Interactions	Communication Partners
C_1	20	C_2, C_3, C_4
C_2	18	C_1, C_3, C_5
C_3	18	C_1, C_2, C_6
C_4	16	C_1, C_5, C_6
C_5	12	C_2, C_4
C_6	12	C_3, C_4

Fig. 8. Dependencies and number of interactions between components $C_1, C_2, C_3, C_4, C_5, C_6$

This is not a wrong result as the following run of the method will provide a configuration where all the components are merged into one and the method ends.

However, this type of result shows that the component configuration contains a high number of interdependencies that are best avoided if all the components are merged. Our method is intended for analysis purposes and by running it several times we may decide that perhaps the initial configuration with $max_1 = 20$ was more balanced than the second one with $max_2 = 24$. Obviously, the method is highly dependent on the maximum desired number of component interdependencies, n .

The Desired Number of Component Interdependencies. The desired number of component interdependencies is relative to each application. In some cases, it can depend on some standards and in some cases, it can depend on the experience of the developers. One can also start from an arbitrary number n and check the generated component configuration, given the initial configuration and such n . If the configuration is not suitable with the purposes of the application, then a higher interdependency level might be tested.

The ProB animations are instrumental in our method as they allow us to determine significant numbers related to component communication. One can see from figures such as Fig. 3 that DecodeUnit communicates with five other components; however, it is not clear how much communication actually occurs on these five ‘channels’.

Another interesting application of our method refers to placing components on hardware platforms so that the highly communicating components are placed close to each other. We study this particular problem in more detail in the next section.

4 Applying the Synthesis Method to NoC Mapping

Recently the Network-on-Chip (NoC) communication paradigm [12] has been proposed as the intercommunication scheme of the cores in Multi-Processor Systems-on-Chip (MPSoC) offering sufficient bandwidth for concurrent on-chip data transactions, incremental scalability of the network, as well as distributed and flexible routing of data.

The mapping of the components of an application to NoC platforms and their subsequent communication patterns contribute essentially to the efficient execution of the application as well as to the performance and power consumption of the NoC system. Meeting these *efficiency* constraints for various design requirements becomes a challenge due to the growing complexity of systems [13]. To address this, application mapping techniques have been recently proposed [14] with the main advantage of reducing the length of the global interconnect which in turn reduces power consumption considerably. In other words, the main contribution of application mapping techniques is to place communicating components as close as possible to each other such that the energy required to transfer data between them is minimised. Hence, the mapping of an application to a NoC platform considerably influences the efficiency of a NoC regardless of the nature of the communication patterns.

We base our synthesis method for efficient application mapping on several assumptions. First, we assume that some NoC communication topology is available for the actual propagation of data between the components of an application. Second, we assume that a certain application can be first generically developed and then "applied" to various network settings. Hence, it is not necessary to consider the network mechanisms when designing the application, but rather have them integrated at a later stage. This separation of concerns is important for reusing the generic application and port it on various platforms. We also assume that neighbouring tiles on a NoC communicate faster than non-neighbouring tiles, hence our goal is to place components that communicate heavily as close as possible to each other.

Based on these assumptions, given an already specified application, we explore the mapping of the application's software components to NoC tiles with the ProB and GeneSyst formal approach to handle and analyse the dependencies between the components. A simplified 2D mesh NoC topology is illustrated in Fig. 9. The dots represent the NoC routers and the white squares represent processing cores as well as storage elements connected to their routers through network interfaces. The interfaces are represented by the black squares within the white squares.

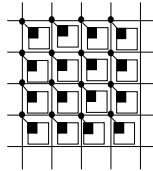


Fig. 9. A 2D NoC mesh

We adapt our synthesis method described in Section 3 to mapping components onto a NoC platform. In Step 4 in our method, instead of defining the new component, we simply place the communicating components as close as possible to the highly communicating one. Moreover, we incorporate the additional ProB facilities to help understand the pipelined traces as intermediate steps in our method, as well as we employ the GeneSyst tool for facilitating the placement in the last step. The steps of our generic method of Section 3 are therefore modified to:

1. We run a (large enough) animations in ProB based on the ProB model of the application.
2. We generate an abstract state-transition diagram with GeneSyst based on specific states of the application.
3. Based on the computed coverage of the model by ProB, we record the relations between the operations of the model's components given in the lists DEFINITELY_ENABLED_AFTER and POSSIBLY_ENABLED.

4. We enrich the operation traces with information about interleaving operations by:
 - (a) observing the history window of ProB for the generated operation traces.
 - (b) observing the computed coverage and the S-MRS graph of the model to understand the different starting points of the interleaving.
5. We replace each operation in the recorded lists of Step 3 with the name of its component.
6. We count the number of occurrences of the name of each component in the table created in Step 5 and store this information in another table. If the same relation occurs in both lists `DEFINITELY_ENABLED_AFTER` and `POSSIBLY_ENABLED`, count it only once. We place the components in the table in descending order of the number of their occurrences.
7. We identify the component that has communication with the biggest number of other components by observing the number associated with it in the previous step and place that component in the 2D NoC mesh at a position where there are enough free tiles around it for the placement of the other components.
8. We parse the rest of the table created in Step 6 and place each remaining unplaced component on the NoC according to the relationships with the other units stored in the table created in Step 6, the interactions shown in the enriched trace created in Step 4 and the information observable by the abstract state-transition diagram generated by GeneSyst in Step 2.
 - (a) If there are components interfering only with one other component, we place them beside the interfering component and out of the way of communication between the other components.
 - (b) We perform the placing of the rest of the components according to the interactions of their operations in relation to the operations of the placed components.

We exemplify our mapping method based on the specification of `PipelinedProcessor` in ProB. We apply our synthesis method starting with the animation of the ProB model (Step 1). In Fig. 5 we observe the history of executed operations of the components.

In Fig. 10 we present the generated abstract state-transition diagram by GeneSyst (Step 2) based on specific states of the application. In this case we focus on the predicate $(fd = \text{FALSE} \ \& \ hz = \text{FALSE})$ given in an additional **ASSERTIONS**-clause in the specification of `PipelinedProcessor` presented in Fig. 4. For each state one can observe the transitions that do not change it and the transitions being able to fire between the states. The given predicate in Fig. 10 corresponds to a single state, hence, we do not show any example of transitions between states.

The condition $fd = \text{FALSE} \ \& \ hz = \text{FALSE}$ is equivalent to the case when a new instruction can be fetched from `InstructionMemoryUnit` and a current instruction can be manipulated in `ExecutionUnit` and be further written to the memory without a possibility for a pipeline hazard. This is illustrated by the group of enabled operations which do not update the state. Operations `Decode1` and `Fetch2` of `DecodeUnit` and `FetchUnit`, respectively, are not included because they update the

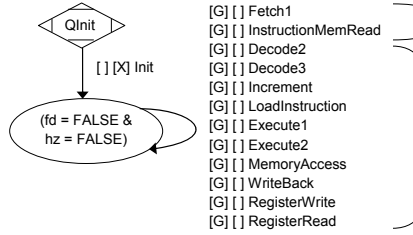


Fig. 10. Abstract state-transition diagram of PipelinedProcessor observing a specific state; (fd = FALSE & hz = FALSE)

value of fd and hz. Operation Hazard of HazardUnit is not included, too, because it cannot be enabled in that state.

After a large enough animation to obtain full state-space exploration, ProB computes the *coverage* of the model (Step 3). Fig. 6 shows part of the information observable from the *computed coverage* which is relevant for our analysis: the list of DEFINITELY_ENABLED_AFTER operations and the list of POSSIBLY_ENABLED operations.

Examination of the S-MRS graph of a model generated by ProB after an animation is instrumental (Step 4) in order to understand the history of executed operations in Fig. 5. Part of the S-MRS graph of the model generated by ProB is shown in Fig. 11.

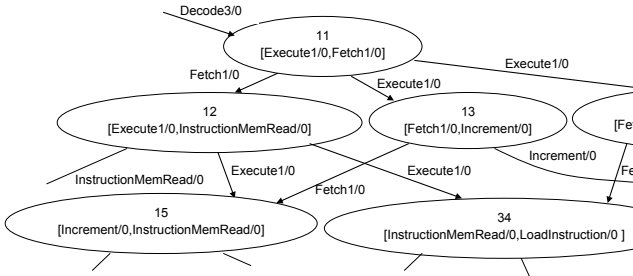


Fig. 11. Part of Signature-Merge Reduced Statespace graph of PipelinedProcessor

From this graph we can observe the different starting points of possible parallel pipelining. For instance, it can be seen that pipelined fetching of a new instruction with operation Fetch1 of FetchUnit may occur at the point where operation Decode3 of DecodeUnit has been executed, or it may occur at the point where operation Execute1 of ExecutionUnit has been executed.

We continue our analysis by creating a trace of the executed operations showing the pipelining and interleaving (Step 4). The history window, the information from the computed coverage and the exploiting of the S-MRS graph of the model are used for understanding the pipelining and interleaving. In Fig. 12 we show such an enriched trace.



Fig. 12. Part of an execution trace of PipelinedProcessor

Operation Fetch1 of FetchUnit initiates a new instruction manipulation and operation RegisterWrite of RegisterAccessUnit completes it, which is shown with the operation names being in bold face. The pipelined manipulation of a new instruction is shown with the operations in parenthesis. For instance, the second execution of operation Fetch1 (the fetching of the second instruction) occurs between operations Decode3 and Execute1 of DecodeUnit and ExecutionUnit, respectively, operating on the first instruction, while the next pipelined fetching of new instructions occur between different pairs of units. A diagrammatic view of part of the trace (lines 2, 3 and 4 in Fig. 12) is illustrated in Fig. 13.

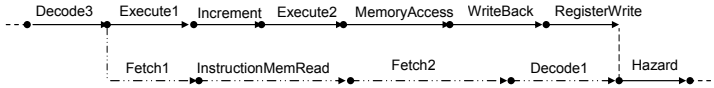


Fig. 13. Diagrammatic view of part of the trace in Fig. 12

The next step of our method (Step 5) consists of replacing each operation in the recorded lists of Step 3 with the name of its component, which can be observed in Fig. 6. We then count the number of occurrences of the name of each component (Step 6) in the table created in Step 5 and store this information in another table (left and central column in Fig. 7).

We continue the analysis and conclude our mapping of the processor components to the NoC 2D mesh shown in Fig. 14. We note that any NoC communication topology can be considered, but in this paper we assumed a 2D mesh as the target topology. The proposed mapping is justified as follows, by performing Steps 7 and 8 of our method.

We first study the table illustrated in Fig. 7 and we identify DecodeUnit as the component that has communication with the biggest number of other components (Step 7) by observing the number associated with it. DecodeUnit is placed on the 2D NoC mesh at a position where there are enough free tiles around it for the placement of the other components.

We parse repeatedly the rest of the table included in Fig. 7 (Step 8) and place ExecutionUnit beside DecodeUnit with a distance between them of one hop, i.e. one connecting line between their corresponding routers shown in Fig. 14. The reason for this placement is because these components communicate regularly according

to the dependencies and pipelining shown in Fig. 13, 12 and 11. Moreover, they both communicate with several non-communicating components that do not need to be placed close together. The third most communicating component is `RegisterAccessUnit` and is placed beside `DecodeUnit` because they communicate according to the interactions shown in Fig. 13, 12 and 11. `RegisterAccessUnit` does not interact with `FetchUnit` as shown in the figures of the previous analysis steps, thus they do not need to be placed close to each other.

Based on the results shown in Fig. 11, 12 and 13, we continue with the placement of `ProgramCounter`. It interferes only with `DecodeUnit` and `ExecutionUnit` which results in placing these three components close together and `ProgramCounter` to be out of the way of the communication between `DecodeUnit`, `ExecutionUnit` and the rest of the components. In a similar manner we can decide on the placement of `HazardUnit`. Based on the previous analysis it communicates with `DecodeUnit` and always allows `DecodeUnit` to continue computation after operation `WriteBack` of `WriteBackUnit` and the update of `RegisterAccessUnit`. `MemoryAccessUnit` communicates with `ExecutionUnit` and `WriteBackUnit`, which in turn communicates with `HazardUnit`. Thus, `MemoryAccessUnit`, `ExecutionUnit`, `WriteBackUnit` and `HazardUnit` need to be placed close together. `InstructionMemoryUnit` interacts only with `FetchUnit`, so it is placed beside `FetchUnit` without interfering with the other units. This is justified too by what is shown in the abstract state-transition diagram generated by GeneSyst in Fig. 10. Considering all these interactions which are based on the analysis in Section 4, we decide on the final placement shown in Fig. 14.

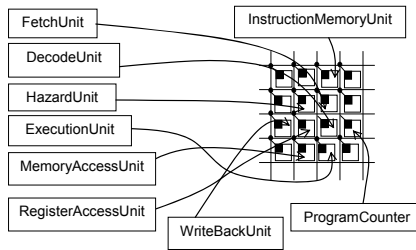


Fig. 14. Placement of the subsystems of `PipelinedProcessor` on a region of a NoC 2D mesh

We can justify the proposed placement of the components of `PipelinedProcessor` based on one of the main metrics for efficient application mapping in the literature, namely *total communication cost*. This metric is calculated by summing all the products of required bandwidth and the distance in hops on the NoC for each pair-wise communication. If we assume that the required bandwidth for each pair-wise communication is equal then the decided placement incurs the minimum total communication cost for the execution of the application. If the bandwidth required for each pair-wise communication was known then this would form one more input to our method. This will be considered in future applications of our method.

5 Related Work

Several frameworks have been developed in order to elevate the representation and analysis of component interdependencies to a separate design problem, orthogonal to the specification of the functional components of an application. We consider here two of them that are closely related to our approach. Cadena [15] is an integrated environment for modeling systems built using the CORBA Component Model (CCM). Cadena provides facilities for defining component types, assembling systems from CCM components, visualizing several dependence relationships between components, specifying and verifying correctness (through the Bogor model checking framework) properties of models of CCM systems, and producing CORBA stubs and skeletons implemented in Java.

Synthesis [11] is a component-based software development environment providing software interconnection dependencies and sets of alternative protocols for managing them. The environment consists of three elements: a software architecture description language called Synopsis, a design handbook of dependencies and associated coordination protocols and a design assistant which generates executable applications by successive specialisations of their Synopsis description.

Several works have been presented during the last years to address the application mapping to 2D NoC designs. The reader is referred to the survey done by Marculescu et al. [13] for a detailed list of them. Specific to pipelined applications, Yang et al. [16] propose a mapping strategy for block cipher security algorithms decomposed into tasks. Each task is mapped to a processing element of a NoC platform. The proposed approach is simulated by a cycle-accurate SystemC model platform called Networked Processor Array. The algorithms are written in C and are then profiled to identify pipelined execution groups which can be performed concurrently or sequentially. The scheduling and mapping step has as inputs the results from the profiling step as well as the available number of processing elements on the NoC platform.

Branca et al. [17] compared four search algorithms for the mapping of pipelined applications on multiprocessor platforms. Thus, Tabu Search, Simulated Annealing, Genetic Algorithms and the Bayesian Optimisation Algorithm were applied on two pipelined applications. The authors concluded that the deployment of the Bayesian Optimisation algorithm was the most efficient.

Zheng et al. [18] proposed a pipelined scheme for implementation of H.264 CABAC decoding. After analysing the original H.264 specification document and employing a finite state machine the authors identified a fixed and steady pipelined behavior to improve the efficiency of the decoder. In contrast to this approach, we provide an analysis method that can assist the user to identify parallel pipelines.

McEwan and Schneider [19] presented modelling and analysis of the AMBA synchronous bus using ProB as the main tool. They stated that ProB is effective in supporting the construction of the formal model at the point it is being developed. In this paper we presented an additional synthesis method within ProB for a later design phase after the formal modelling and verification of a software application has taken place.

6 Conclusions

Formal methods with adequate tool support are important for the design and analysis of complex systems in order to correct errors in the early design phases and reduce the involved costs of system design and development.

In this paper we propose a novel method for analyzing component boundaries, based on a desired interdependency level between component services. The component interdependencies are determined based on ProB animation facilities such as the definite and the possible communications for one component. This is a decision we assume for our method here, but elsewhere [20] we have devised a method based only on the definite communications. We demonstrate the suitability of our proposed method to efficient application mapping to a NoC platform, which we believe is one potential application domain of our method.

To the best of our knowledge, this is the first approach to use model checking and animating tools of formal methods in order to facilitate the analysis of component interdependencies. This development is thus important in order to design dependable systems.

Specific to applying our analysis method for facilitating efficient application mappings to NoC platforms, in this paper we did not consider the actual computation volume of each subcomponent, nor the required bandwidth for the component intercommunication which affects the total communication cost for the mapped application. These are additional metrics for deciding the mapping of an application to NoC platforms and will be considered in future applications of our method.

Acknowledgement. This work is supported by IST FP7 DEPLOY project. The authors would like to thank the anonymous reviewers of this paper for their helpful comments.

References

1. Dellarocas, C.: The SYNTHESIS Environment for Component-Based Software Development. In: 8th International Workshop on Software Technology and Engineering Practice (STEP 1997), p. 434 (1997)
2. Szyperski, C.: Component Software - Beyond Object-Oriented Programming. Addison-Wesley, ACM Press (1998)
3. Lecomte, T.: Safe and Reliable Metro Platform Screen Doors Control/Command Systems. In: Cuellar, J., Sere, K. (eds.) FM 2008. LNCS, vol. 5014, pp. 430–434. Springer, Heidelberg (2008)
4. Bernardo, M., Cimatti, A. (eds.): SFM 2006. LNCS, vol. 3965. Springer, Heidelberg (2006)
5. Waldén, M., Sere, K.: Reasoning about Action Systems using the B Method. Formal methods in System Design 13(1), 5–35 (1998)
6. Abrial, J.-R.: The B-Book. Cambridge University Press, Cambridge (1996)
7. Leuschel, M., Butler, M.: ProB: A Model Checker for B. In: International Symposium of Formal Methods Europe, pp. 855–874 (2003)

8. Bert, D., Potet, M.-L., Stouls, N.: GeneSyst: A Tool to Reason About Behavioral Aspects of B Event Specifications, Application to Security Properties. In: Treharne, H., King, S., C. Henson, M., Schneider, S. (eds.) ZB 2005. LNCS, vol. 3455, pp. 299–318. Springer, Heidelberg (2005)
9. Back, R.J.R., Kurki-Suonio, R.: Decentralization of Process Nets with Centralized Control. In: The 2nd Symposium on Principles on Distributed Computing, pp. 131–142 (1983)
10. SICS, SICStus Prolog (2006), website: <http://www.sics.se/sicstus>
11. Plosila, J., Sere, K.: Action Systems in Pipelined Processor Design. In: 3rd International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC 1997), p. 156 (1997)
12. Hemani, A., Jantch, A., Kumar, K., Postula, A., Öberg, J., Millberg, M., Lindqvist, D.: Network on a Chip: An architecture for billion transistor era. In: IEEE NorChip Conference (2000)
13. Marculescu, R., Ogras, U., Peh, L.-S., Enright Jerger, N., Hoskote, Y.: Outstanding Research Problems in NoC Design: System, Microarchitecture, and Circuit Perspectives. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 28(1), 3–21 (2009)
14. Hu, J., Marculescu, R.: Energy-aware mapping for tile-based NOC architectures under performance constraints. In: Asia South Pacific Design Automation Conference, Japan, pp. 233–239 (2003)
15. Childs, A., Greenwald, J., Ranganath, V.P., Deng, X., Dwyer, M.B., Hatcliff, J., Jung, G., Shanti, P., Singh, G.: Cadena: An Integrated Development Environment for Analysis, Synthesis, and Verification of Component-Based Systems. In: Wermelinger, M., Margaria-Steffen, T. (eds.) FASE 2004. LNCS, vol. 2984, pp. 160–164. Springer, Heidelberg (2004)
16. Yang, Y.S., Bahn, J.H., Lee, S.E., Bagherzadeh, N.: Parallel and Pipeline Processing for Block Cipher Algorithms on a Network-on-Chip. In: Sixth International Conference on Information Technology: New Generations, pp. 849–854 (2009)
17. Branca, M., Camerini, L., Ferrandi, F., Lanzi, P.L., Pilato, C., Sciuto, D., Tumeo, A.: Evolutionary Algorithms for the Mapping of Pipelined Applications onto heterogeneous Embedded Systems. In: 11th Annual Conference on Genetic and Evolutionary Computation, GECCO 2009 (2009)
18. Zheng, J., Wu, D., Xie, D., Gao, W.: A novel pipeline design for H.264 CABAC decoding. In: Ip, H.H.-S., Au, O.C., Leung, H., Sun, M.-T., Ma, W.-Y., Hu, S.-M. (eds.) PCM 2007. LNCS, vol. 4810, pp. 559–568. Springer, Heidelberg (2007)
19. McEwan, A.A., Schneider, S.: Modeling and analysis of the AMBA bus using CSP and B. In: Communicating Process Architectures, pp. 379–398 (2007)
20. Petre, L., Sere, K., Tsiopoulos, L., Liljeberg, P., Plosila, J.: Towards Self-Placing Applications on 2D- and 3D-NoCs. In: Cong-Vinh, P. (ed.) *Autonomic Networking-on-Chip: Bio-inspired Specification, Development, and Verification, Embedded Multi-core System(EMS) Book Series*. CRC Press, Boca Raton (to appear, 2011)

Shared Event Composition/Decomposition in Event-B^{*}

Renato Silva^{**} and Michael Butler

School of Electronics and Computer Science
University of Southampton, UK
{ras07r,mjb}@ecs.soton.ac.uk

Abstract. The construction of specifications is often a combination of smaller sub-components. *Composition* and *decomposition* are techniques supporting reuse and allowing formal combination of sub-components through refinement steps. Sub-components can result from a design or architectural goal and a refinement framework should allow them to be further developed, possibly in parallel. We propose the definition of composition and decomposition in the Event-B formalism following a shared event approach where sub-components interact via synchronised shared events and shared states are not allowed. We define the necessary proof obligations to ensure valid compositions and decompositions. We also show that shared event composition preserves refinement proofs, that is, in order to maintain refinement of compositions, it is sufficient to prove refinement between corresponding sub-components. A case study applying these two techniques is illustrated using Rodin, the Event-B toolset.

Keywords: formal methods, composition, decomposition, reuse, Event-B, design techniques, specification.

1 Introduction

The development of specifications in a “top-down” style starts with an abstract model of the envisaged system. Systems can often be seen as a combination and interaction of several sub-specifications (hereafter called sub-components) where each sub-component has its own functionality aspect. This view introduces *modularity* in the system: different sub-components represent a particular functionality and changes in the sub-components are accommodated more gracefully [1] in the system specification. We use *composition* to structure specifications through the interaction of sub-components seen as independent modules. This use of composition is not new in other formal notations: examples are [2,3,4]. Here we express how we can use (and reuse) composition for building specifications

* Part of this research was carried out within the European Commission ICT project 214158 DEPLOY (<http://www.deploy-project.eu>)

** R. Silva receives a Doctoral Degree Grant sponsored by Fundação Ciência e Tecnologia (FCT-Portugal).

in Event-B [5] through the interaction of sub-components (modules), benefiting from their properties and proof obligations (POs). The interesting part of composition involves the interaction of sub-components which usually occurs by means of shared state [6], shared operations [7] or a combination of both (for example, fusion composition [4]). In CSP [8,9] shared actions labels can be synchronised. We take a similar approach in Event-B and we *synchronise events* independently of their labels in a *shared event composition approach*. Properties of the CSP synchronisation such as monotonicity remain valid for the shared event composition. Butler [7] using Action Systems [10] and Classical B [11] defines the parallel composition of action systems including parallel composition with value-passing. We follow this approach to define the shared event composition for Event-B.

Decomposition is motivated by the possibility of breaking a complex problem or system into parts that are easier to conceive, manage and maintain. The partition of a model into sub-components can also be seen as a design/architectural decision and the further development of the sub-components in parallel is possible. Besides alleviating the complexity for large systems and the respective proofs, decomposition allows team development in parallel over the same model which is very attractive in an industrial environment. Moreover the proof obligations of the original (non-decomposed) model can be reused by the sub-components. The proof obligations to ensure a valid composition are expressed including the possibility to reuse the sub-components properties. We present in more detail the shared event approach applied to composition and decomposition. The monotonicity property for composition is proved by means of refinement proof obligations. We see decomposition as the inverse operation of composition and therefore we can reuse its properties to decompose systems. Guidelines for applying a shared event decomposition are presented illustrated by a case study. The models are developed in Rodin [12], an Event-B toolset [5,13].

This document is structured as follows: Section 2 gives an overview of the Event-B formal method. Section 3 introduces the notion and motivation for the shared event approach for composition and decomposition. Composed machines, properties, proof obligations are described in Sect. 4. Decomposition guidelines are presented in Sect. 5. Section 6 illustrates the application of composition and decomposition to a distributed system case study: file access system. Related work is described in Sect. 7. Conclusions and future work are drawn in Sect. 8.

2 Event-B Language

Event-B, inspired by Action Systems, Classical B and Z [14], is a formal modelling method for developing *correct-by-construction* hardware and software systems. An Event-B model is a state transition system where the state corresponds to a set of *variables* v and transitions are represented by *events*. Essential is the formulation of *invariants* $I(v)$: safety conditions to be preserved at all times.

An abstract Event-B specification is divided into a static part called *context* and a dynamic part called *machine* as seen in Fig. 1. A context consists of sets s

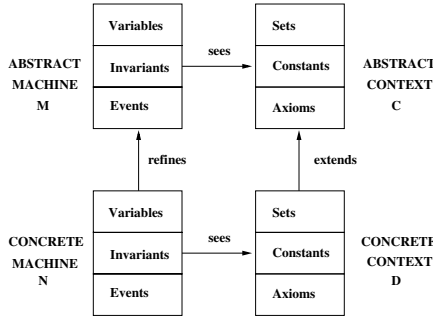


Fig. 1. Machine and context refinement

(collection of elements or type definitions), constants c and axioms $A(\dots)$ of the system. A machine contains the state (global) variables v whose values are assigned in *events*. Events, that can be parameterised by local variables p , occur when their conditions (called *guards* $G(\dots)$) are true and as a result the state variables may be updated by *actions* $S(\dots)$. *Invariants* $I(\dots)$ define the dynamic properties of the specification and POs are generated to verify that these properties are always maintained. The most general form of an event is

$$e \hat{=} \mathbf{ANY} \ p \ \mathbf{WHERE} \ G(s, c, p, v) \ \mathbf{THEN} \ S(s, c, p, v, v') \ \mathbf{END}.$$

where event e is expressed by parameters p , guards $G(s, c, p, v)$ and actions $S(s, c, p, v, v')$. When guard $G(s, c, p, v)$ is true then event e is enabled and therefore the action $S(s, c, p, v, v')$ updates the set of variables v to v' (value of v after the assignment).

To facilitate the construction of large-scale models, Event-B advocates the use of *refinement*: the process of gradually adding details to a model. Refinement of a machine consists of refining existing events. An Event-B development is a sequence of models linked by refinement relations. It is said that a concrete model refines an abstract one. Abstract variables v are linked to concrete variables w by a *gluing invariant* $J(v, w)$. POs are generated to ensure that this invariant is preserved in the concrete model. Any behaviour of the concrete model must be *simulated* by some behaviour of the abstract model, with respect to the gluing invariant $J(v, w)$. New events can be added, refining *skip* which may be declared as convergent, meaning they do not cause divergence. The convergence is proved if each new event decreases a *variant*. The variant must be well-founded and may be an integer or a finite set.

3 Shared Event Approach

The shared event approach is suitable for the development of distributed systems [7]: sub-components interact through synchronised events in parallel.

¹ (...) refers to the free identifiers in the expression like sets, constants, etc.

In CSP, synchronised input or output channels can exchange messages. In Event-B, the sub-component events can exchange messages via *shared parameters* which is useful for modelling message broadcasting systems. Next we describe how we define a shared event composition in Event-B.

3.1 Shared Event Composition

Sub-component specifications that are part of a full system specification deal with a particular part of the system being modelled. Sub-component interaction must be verified to comply with the desired behavioural semantics of the system. We focus on developments using shared event composition where individual elements' properties are conjoined: *conjunction* of individual invariants, *conjoining* variables and *synchronisation* of events.

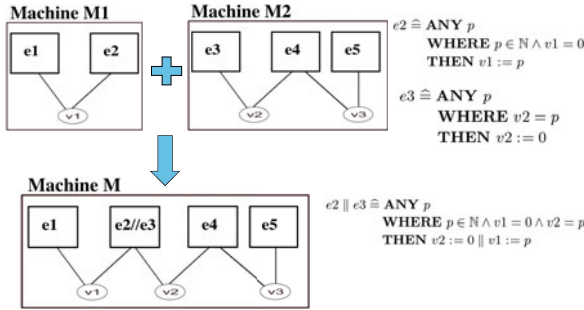


Fig. 2. Shared event composition of $M1$ and $M2$ (a) resulting in M (b)

Consider Fig. 2 where machine $M1$ has events $e1$ and $e2$ using variable $v1$. Moreover machine $M2$ has events $e3$, $e4$ and $e5$ using variables $v2$ and $v3$. Events $e2$ and $e3$ can occur in parallel (independent variables) and can be synchronised. In Fig. 2, machine M is the result of the shared event composition of machines $M1$ and $M2$ where $e2$ from machine $M1$ and $e3$ from machine $M3$ are composed: $e2 \parallel e3$. The interaction of machines $M1$ and $M2$ through their events results in a *composed event* sharing two independent variables: $v1$ and $v2$.

Butler [7] defines a general definition for the parallel composition of action systems with value-passing fusion. Based on that work, we can express a general definition for the parallel composition of generic events e_a and e_b as Def. 1:

Definition 1. *Composition of events e_a and e_b with a common parameter p results in:*

$$\begin{aligned}
 e_a &\hat{=} \text{ANY } p?, x \text{ WHERE } G(p?, x, m) \text{ THEN } S(p?, x, m) \text{ END} \\
 e_b &\hat{=} \text{ANY } p!, y \text{ WHERE } H(p!, y, n) \text{ THEN } T(p!, y, n) \text{ END} \\
 e_a \parallel e_b &\hat{=} \text{ANY } p!, x, y \text{ WHERE } G(p!, x, m) \wedge H(p!, y, n) \\
 &\text{ THEN } S(p!, x, m) \parallel T(p!, y, n) \text{ END}
 \end{aligned}$$

where x, y, p are sets of parameters from each of the events e_a and e_b . Event e_a has $p?$ as an input parameter and e_b has $p!$ as an output parameter and the resulting composition is $p!$ itself an output parameter, modelling the passing of the output value from the output parameter to the input parameter. This property can be used to model value-passing systems: e_b sends a value to e_a using the common parameter p . Communication between input type parameters is also possible but not for both output parameters since the output parameters may not be willing to output the same value, leading to a deadlock state. Although it is possible to compose events e_a and e_b even if they share variables, this would lead to a shared variable decomposition which out of the scope of this document since we focus on the shared event decomposition that restricts variable sharing. More information about that kind of composition can be found in [6].

Action systems [10] provide a general description of reactive systems, capable of modelling terminating, aborting and infinitely repeating systems. Event-B is inspired by action systems and can be seen as a realisation of actions systems but using a combination of logic and mathematics. Both formalisms share the same refinement semantics. Therefore we claim that Event-B has the same semantic structure and refinement definitions as action systems. It is possible to make a correspondence between parallel composition in CSP and an event-based view of parallel composition for action systems [15,16].

Theorem 1. *The shared event parallel composition of actions systems corresponds to the CSP parallel composition. The failure-divergence semantics of CSP can be applied to action systems. The failure divergence semantics of action system M in parallel with N , $M \parallel N$ is defined as:*

$$\llbracket M \parallel N \rrbracket = \llbracket M \rrbracket \parallel \llbracket N \rrbracket$$

where $\llbracket M \rrbracket$ and $\llbracket N \rrbracket$ are the failure divergence semantics of M and N respectively. The proof of this theorem can be found in [15].

The semantics of the parallel composition of machines M and N corresponds to the set of failure-divergence for each individual machine in parallel. The parallel operator for value-passing action-systems enjoys properties such as monotonicity and associativity [15]. There is a correspondence between action systems and Event-B. Action system is a predicate transformer from a precondition P to post-condition Q with variables v possibly being modified. Event-B events are similar but from a more specific view where the guards correspond to preconditions P , actions R correspond to post-condition Q and the same variables v are possibly modified:

$$[\text{ANY } x \text{ WHERE } P(x, v) \text{ THEN } v : | R(x, v, v') \text{ END}]Q$$

An action in action systems is expressed by:

$$(\forall x \cdot P(x, v) \Rightarrow [v : | R(x, v, v')]Q) \Leftrightarrow (\forall x \cdot P(x, v) \Rightarrow (\forall v \cdot R(x, v, v') \Rightarrow Q))$$

Event-B can be seen as a realisation of the generic action system formalism where there is a direct correspondence between Action System actions and Event-B

events. From the correspondence between action systems and Event-B, machines M and N can be refined independently which is one of the most important and powerful properties that shared event composition in Event-B inherits from CSP. The monotonicity property for the shared event composition in Event-B is proved by means of proof obligation in Sect. 4.3. An advantage of using Event-B is the tool support available through the Rodin platform where proof obligations are automatically generated.

When sub-components are composed it is desirable to define properties that relate the individual sub-components allowing interactions. These properties are expressed by adding *composition invariants* $I_{CM}(s, c, v_1, \dots, v_m)$ to the composed machine constraining the variables of all machines being composed.

Definition 2. *The invariant of the parallel composition of machines M_1 to M_m with variables v_1 to v_m respectively is the conjunction of the individual invariants and the composition invariant $I_{CM}(s, c, v_1, \dots, v_m)$:*

$$I(M_1 \parallel \dots \parallel M_m) \hat{=} I_1(s, c, v_1) \wedge \dots \wedge I_m(s, c, v_m) \wedge I_{CM}(s, c, v_1, \dots, v_m). \quad (1)$$

In Fig. 2, *composed machine* M has as invariant the conjunction of the individual invariants $I(A \parallel B) \hat{=} I_A(s, c, v_1) \wedge I_B(s, c, v_2, v_3)$ plus a possible composition invariant $I_{CM}(s, c, v_1, v_2, v_3)$. In a shared event composition the sub-components have independent state space (variables are unique to each machine). Consequently, composition reasoning is simplified, as there are no constraints between state spaces of sub-components.

3.2 Shared Event Decomposition

Decomposition can be seen as the inverse process of composition: after some refinements a larger model may be decomposed into smaller components. This step might be a consequence of complexity or just as an architectural decision. The shared event approach is also used: *events are shared* between sub-components and *variable sharing is not allowed*. Butler [17] proposes a shared event decomposition for Event-B inspired by CSP and action systems with event sharing as seen in Fig. 3. We follow that work in our approach.

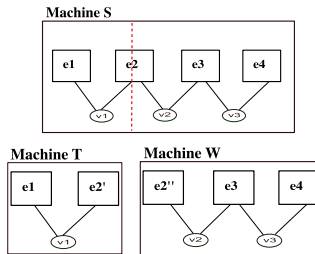


Fig. 3. Shared event decomposition of machine S into T and W sharing e_2

The decomposition is obtained by selecting which variables from the original model are allocated to which sub-component. Therefore, events using variables allocated to different sub-components ($e2$ shares $v1$ and $v2$) must be split (described in Sect. 5). The part corresponding to each variable ($e2'$ and $e2''$) is used to create partial versions of the original event. After the decomposition, the individual machines can be further refined since the composition relation holds. The possible recomposition of the sub-components (or their refinements) is a refinement of the original composed component although this step should never be required in practice.

4 Composed Machines: Composition and Refinement

We define a new construct *composed machine*, representing the shared event composition of Event-B machines. We aim to have a construct that remains reactive to changes in the sub-components. Consequently the composition is *structural*. The interaction of sub-components follows a “top-down” approach, representing a *refinement* of an existing abstraction. To formalise the composition, it is necessary to define composition and refinement POs. In the following sections, we introduce the structure of a composed machine, respective POs and prove the monotonicity property.

4.1 Structure of Composed Machines

A shared event composed machine is expressed as the parallel conjunction of machines. Machines are composed in parallel including their invariants, variables and events: $CM \hat{=} M1 \parallel \dots \parallel Mm$ as seen in Fig. 4. Moreover:

- The composed machine variables are all the sub-component variables (v_1 from $M1$, v_2 from $M2$, \dots , v_m from Mm) and are state-space disjoint.
- The invariants of the composed machine are defined as Def. 2.
- The composed events are defined according to Def. 1.

<pre> COMPOSED MACHINE CM INCLUDES M_1, \dots, M_m VARIABLES v_1, \dots, v_m INVARIANTS $I_{CM}(s, c, v_1, v_2, \dots, v_m)$ EVENTS $e_{11} \hat{=} M1.e_{11} \parallel \dots Mm.e_{m1}$ \dots $e_{1p} \hat{=} M1.e_{1p} \parallel \dots Mm.e_{m1} \ e_{1p}$ END </pre>
--

Fig. 4. Composed machine CM composing $M1$ to Mm and seeing context Ctx

When a composed machine is used as a combination of composition and refinement, it refines an abstract model and just like in an ordinary machine, abstract events must be refined. For instance, a composed machine CM resulting from the parallel composition of $M1 \dots Mm$ and refining abstract machine $M0$ can be expressed as $M0 \sqsubseteq CM \equiv M0 \sqsubseteq M1 \parallel \dots \parallel Mm$. Next we present the required POs to verify composed machines.

4.2 Proof Obligations

POs play an important role in Event-B developments. POs are generated to verify the properties of a model. For simplicity we define POs in terms of a composition of two machines M_1 and M_2 that refine machine M_0 , but the rules generalise easily to the composition of n machines. Furthermore context elements such as sets, constants and axioms ($s, c, A(s, c)$) that are part of the static side of a specification, are not considered in the formulas. The POs defined for standard machines are [5]:

- Consistency: Invariant Preservation (INV) and Feasibility (FIS)
- Refinement: Guard Strengthening (GRD), Simulation/Refinement (SIM) and Gluing Invariant Preservation (INV)
- Variant: Numeric Variant (NAT), Numeric Variant Decreasing (VAR), Finite Set Variant (FIN)
- Well-Definedness(WD)

Invariant Preservation and Gluing Invariant Preservation POs differ in that the first refers to the invariant in the abstract machine while the second refers to invariant relating abstract and concrete variables in a (concrete) refinement machine. These POs also are defined for composed machines except the ones related with variant (no variant for composed machines). We simplify the composed machines POs by assuming that the POs of the individual machines hold. We explain and define the additional POs necessary to ensure that the composed machine satisfies all the standard POs. We consider that the POs of M_0 , M_1 and M_2 hold. The respective composition POs are described as follows.

Consistency. Consistency POs are required to be always verified. The feasibility proof obligation for the composed event $e_1 \parallel e_2$ is $FIS_{e_1 \parallel e_2}$.

The Feasibility PO ensures that each non-deterministic action is feasible for a particular event. The goal is to ensure that values exist for variables v' such that the before-after predicate $S(p, s, c, v, v')$ is feasible.

Theorem 2. *The FIS PO of individual events can be reused for proving the feasibility for each composed event and that is enough to verify this property. The feasibility PO for the composed event $e_1 \parallel e_2$ can be expressed by the feasibility PO of e_1 (FIS_{e_1}) and e_2 (FIS_{e_2}).*

$$FIS_{e_1} : I_1(v_1) \wedge G_1(p_1, v_1) \vdash \exists v'_1 \cdot (S_1(p_1, v_1, v'_1)) \quad (2)$$

$$FIS_{e_2} : I_2(v_2) \wedge G_2(p_2, v_2) \vdash \exists v'_2 \cdot (S_2(p_2, v_2, v'_2)) \quad (3)$$

$$FIS_{e_1 \parallel e_2} : IC_M(v_0, v_1, v_2) \wedge I_1(v_1) \wedge I_2(v_2) \wedge G_1(p_1, v_1) \wedge G_2(p_2, v_2) \vdash \exists v'_1, v'_2 \cdot (S_1(p_1, v_1, v'_1) \wedge S_2(p_2, v_2, v'_2)). \quad (4)$$

Assume: FIS_{e_1} and FIS_{e_2} .

Prove: $FIS_{e_1 \parallel e_2}$.

Proof. Assume the hypotheses of $FIS_{e1 \parallel e2}$.

$$I_{CM}(v_0, v_1, v_2)$$

$$I_1(v_1) \wedge G_1(p_1, v_1) \tag{5}$$

$$I_2(v_2) \wedge G_2(p_2, v_2). \tag{6}$$

Prove: $\exists v'_1, v'_2 \cdot (S_1(p_1, v_1, v'_1) \wedge S_2(p_2, v_2, v'_2))$. The proof proceeds as follows:

$$\begin{aligned} & \exists v'_1, v'_2 \cdot (S_1(p_1, v_1, v'_1) \wedge S_2(p_2, v_2, v'_2)) \\ & \equiv \exists v'_1 \cdot (S_1(p_1, v_1, v'_1)) \wedge \exists v'_2 \cdot (S_2(p_2, v_2, v'_2)) & \{\text{disjoint } v_1 \text{ and } v_2\} \\ \Leftarrow & (FIS_{e1} \wedge FIS_{e2}). & \{\text{\textcircled{2}} + \text{\textcircled{5}}, \text{\textcircled{3}} + \text{\textcircled{6}}\} \end{aligned}$$

Another consistency PO is invariant preservation. In the composed machine, invariant preservation PO INV_{CM} corresponds to the invariant preservation in all events from the individual machines that are composed. The invariant preservation proof obligation for the composed event $e1 \parallel e2$ is $INV_{e1 \parallel e2}$.

Theorem 3. *This kind of proof obligation ensures that each invariant is preserved by each event. The goal is each individual invariant from the set of existing invariants. For each invariant i from the set of invariants I in a composed machine, the composition invariant $I_{CM}(v_0, v_1, v_2)$ needs to be verified.*

$$INV_{e1} : I_1(v_1) \wedge G_1(p_1, v_1) \wedge S_1(p_1, v_1, v'_1) \vdash i_1(v'_1) \tag{7}$$

$$INV_{e2} : I_2(v_2) \wedge G_2(p_2, v_2) \wedge S_2(p_2, v_2, v'_2) \vdash i_2(v'_2) \tag{8}$$

$$\begin{aligned} INV_{e1 \parallel e2} : & I_{CM}(v_0, v_1, v_2) \wedge I_1(v_1) \wedge I_2(v_2) \\ & \wedge G_1(p_1, v_1) \wedge G_2(p_2, v_2) \wedge S_1(p_1, v_1, v'_1) \wedge S_2(p_2, v_2, v'_2) \\ & \vdash i_1(v'_1) \wedge i_2(v'_2) \wedge i_{CM}(v_0, v'_1, v'_2) \end{aligned}$$

Assume: INV_{e1} and INV_{e2} .

Prove: $INV_{e1 \parallel e2}$.

Proof. Assume the hypotheses of $INV_{e1 \parallel e2}$.

$$I_{CM}(v_0, v_1, v_2)$$

$$I_1(v_1) \wedge G_1(p_1, v_1) \wedge S_1(p_1, v_1, v'_1) \tag{9}$$

$$I_2(v_2) \wedge G_2(p_2, v_2) \wedge S_2(p_2, v_2, v'_2) \tag{10}$$

Prove: $i_1(v'_1) \wedge i_2(v'_2) \wedge i_{CM}(v_0, v'_1, v'_2)$. The proof proceeds as follows:

$$\begin{aligned} & i_1(v'_1) \wedge i_2(v'_2) \wedge i_{CM}(v_0, v'_1, v'_2) \\ \Leftarrow & INV_{e1} \wedge INV_{e2} \wedge i_{CM}(v_0, v'_1, v'_2). & \{\text{\textcircled{7}} + \text{\textcircled{9}}, \text{\textcircled{8}} + \text{\textcircled{10}}\} \end{aligned}$$

Well-definedness for expressions (guards, actions, invariants, etc) needs to be verified. These are verified by means of POs in Event-B [18]. For composed machines, well-definedness POs are only generated for $I_{CM}(v_0, v_1, v_2)$. Other expressions are verified in the individual machines.

Refinement. Refinement POs are required when the composed machine refines an abstract machine. Machine M_0 with variables v_0 , invariant $I_0(v_0)$ and abstract event e_0 is refined by composed machine CM defined by machines M_1 with variables w_1 , invariant $I_1(w_1)$, event e_1 and M_2 (w_2 ; $I_2(w_2)$; e_2) and composition invariant $J_{CM}(v_0, w_1, w_2)$. The composed event $e1 \parallel e2$ refines the abstract event e_0 . The refinement PO results from the verification of the invariant preservation $J_M(v_0, w_i)$, the verification of guard strengthening for $G_0(p_0, v_0)$ and simulation $S_0(p_0, v_0, v'_0)$ for each concrete event. A general refinement PO (REF_{ei}) for a machine M refining event ei follows from:

$$\begin{aligned} REF_{ei} &\hat{=} I_i(v_i) \wedge J_i(v_i, w_i) \wedge H_i(q_i, w_i) \wedge T_i(q_i, w_i, w'_i) \\ &\vdash \exists v'_i. G_i(v_i) \wedge S_i(p_i, v_i, v'_i) \wedge J_i(v'_i, w'_i) \end{aligned} \quad (11)$$

Theorem 4. For each composed event $e1 \parallel e2$, refining abstract event $e0$ through (gluing) composition invariant in a composed machine, the refinement REF PO consists in proving the guard strengthening of abstract guards, proving the simulation of the abstract variables (v'_0) and preserving the gluing invariant ($J_{CM}(v'_0, w'_1, w'_2)$). From (11):

$$INV_{e1} : I_1(w_1) \wedge H_1(q_1, w_1) \wedge T_1(q_1, w_1, w'_1) \vdash i_1(w'_1) \quad (12)$$

$$INV_{e2} : I_2(w_2) \wedge H_2(q_2, w_2) \wedge T_2(q_2, w_2, w'_2) \vdash i_2(w'_2) \quad (13)$$

$$\begin{aligned} REF_{e0 \sqsubseteq (e1 \parallel e2)} : & I_0(v_0) \wedge I_1(w_1) \wedge I_2(w_2) \wedge J_{CM}(v_0, w_1, w_2) \\ & \wedge H_1(q_1, w_1) \wedge H_2(q_2, w_2) \wedge T_1(q_1, w_1, w'_1) \wedge T_2(q_2, w_2, w'_2) \\ & \vdash \exists v'_0. G_0(p_0, v_0) \wedge S_0(p_0, v_0, v'_0) \wedge I_1(w'_1) \wedge I_2(w'_2) \wedge J_{CM}(v'_0, w'_1, w'_2) \end{aligned}$$

Assume: INV_{e1} (12) and INV_{e2} (13).

Prove: $REF_{e0 \sqsubseteq (e1 \parallel e2)}$.

Proof. Assume the hypotheses of $REF_{e0 \sqsubseteq (e1 \parallel e2)}$. Prove: $\exists v'_0. G_0(p_0, v_0) \wedge S_0(p_0, v_0, v'_0) \wedge I_1(w'_1) \wedge I_2(w'_2) \wedge J_{CM}(v'_0, w'_1, w'_2)$. The proof proceeds as follows:

$$\begin{aligned} & \exists v'_0. G_0(p_0, v_0) \wedge S_0(p_0, v_0, v'_0) \\ & \wedge I_1(w'_1) \wedge I_2(w'_2) \wedge J_{CM}(v'_0, w'_1, w'_2) \\ \equiv & G_0(p_0, v_0) \wedge I_1(w'_1) \wedge I_2(w'_2) \\ & \wedge \exists v'_0. (S_0(p_0, v_0, v'_0) \wedge J_{CM}(v'_0, w'_1, w'_2)) \quad \{\wedge \text{goal; } v_0, w'_1, w'_2 \text{ are free variables}\} \\ \equiv & G_0(p_0, v_0) \\ & \wedge \exists v'_0. (S_0(p_0, v_0, v'_0) \wedge J_{CM}(v'_0, w'_1, w'_2)) \quad \{\text{from (12) and (13)}\} \end{aligned}$$

These are the required POs to verify composed machines. Next we show that composed machines are monotonic which allows to further refine individual machines preserving composition.

4.3 Monotonicity of Shared Event Composition for Composed Machines

An important property of the shared event composition in Event-B is *monotonicity*. We prove it by means of refinement POs confirming that this property holds

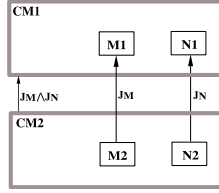


Fig. 5. Refinement of composed machine $CM1 \hat{=} M1 \parallel N1$ by $CM2 \hat{=} M2 \parallel N2$

as it happens for actions systems and CSP described by Butler [15]. Figure 5 shows abstract component specification $M1$ composed with other component specification $N1$, creating a composed model $M1 \parallel N1$. $M1$ is refined by $M2$ and $N1$ by $N2$ respectively. Once we compose specifications $M1$ and $N1$, discharge the required composed POs, $M1$ and $N1$ can be refined individually while the composition properties are preserved without the need to recompose refinements $M2$ and $N2$. We want to formally prove the monotonicity property through refinement POs between composed machines. Therefore if the refinement POs hold between $CM1$ and $CM2$ then $CM1 \sqsubseteq CM2$. Events e_{M1} in machine $M1$ and e_{M2} in machine $M2$ are represented as:

$$e_{M1} \hat{=} \mathbf{ANY} \ p_M \ \mathbf{WHERE} \ G_M(p_M, v_M) \ \mathbf{THEN} \ S_M(p_M, v_M, v'_M) \ \mathbf{END} \quad (14)$$

$$e_{M2} \hat{=} \mathbf{ANY} \ q_M \ \mathbf{WHERE} \ H_M(q_M, w_M) \ \mathbf{THEN} \ T_M(q_M, w_M, w'_M) \ \mathbf{END} \quad (15)$$

The gluing invariant of the refinement between $M1$ and $M2$ is expressed as $J_M(v_M, w_M)$ relating the states of $M1$ and $M2$: $M1 \sqsubseteq_{J_M} M2$. We can derive the refinement PO between $M2$ and $M1$ for the concrete event e_{M2} refining abstract event e_{M1} .

$$\begin{aligned} REF_{e_{M1} \sqsubseteq e_{M2}} : \quad & I_M(v_M) \wedge J_M(v_M, w_M) \wedge G_M(p_M, v_M) \wedge H_M(q_M, w_M) \\ & \wedge S_M(p_M, v_M, v'_M) \wedge T_M(q_M, w_M, w'_M) \\ & \vdash \exists v'_M \cdot G_M(p_M, v_M) \wedge S_M(p_M, v_M, v'_M) \wedge J_M(v'_M, w'_M). \end{aligned} \quad (16)$$

The refinement PO between $N2$ and $N1$ is similar. We refine an abstract event in $CM1$ by a concrete one in $CM2$ and verify that the refinement POs for each individual machine hold for the composition. Event e_{M1} from machine $M1$ and event e_{N1} from machine $N1$ are composed, resulting in the abstract composed event $e_{M1} \parallel e_{N1}$ in $CM1$ from Fig. 5. The gluing invariant relating the states of $CM1$ and $CM2$ is expressed as the conjunction of the gluing invariants between ($M1$ and $M2$) and ($N1$ and $N2$):

$$J_{CM}(v_M, v_N, w_M, w_N) = J_M(v_M, w_M) \wedge J_N(v_N, w_N) \quad (17)$$

Theorem 5. *The refinement POs for composed machines is expressed as the conjunction of the refinement POs for the individual machines. Therefore the monotonicity property holds if the refinement POs of individual machines hold.*

The refinement PO between concrete composed event $e_{M2} \parallel e_{N2}$ and abstract composed event $e_{M1} \parallel e_{N1}$ is expressed as:

$$\begin{aligned}
 REF_{(e_{M1} \parallel e_{N1}) \sqsubseteq (e_{M2} \parallel e_{N2})} : & \quad I_M(v_M) \wedge I_N(v_N) \wedge J_{CM}(v_M, v_N, w_M, w_N) \\
 & \quad \wedge H_M(q_M, w_M) \wedge H_N(q_N, w_N) \\
 & \quad \wedge T_M(q_M, w_M, w'_M) \wedge T_N(q_N, w_N, w'_N) \\
 & \quad \vdash \exists v'_M, v'_N \cdot G_M(p_M, v_M) \wedge G_N(p_N, v_N) \\
 & \quad \wedge S_M(p_M, v_M, v'_M) \wedge S_N(p_N, v_N, v'_N) \\
 & \quad \wedge J_{CM}(v'_M, v'_N, w'_M, w'_N). \tag{18}
 \end{aligned}$$

Assume: $REF_{e_{M1} \sqsubseteq e_{M2}}$ and $REF_{e_{N1} \sqsubseteq e_{N2}}$.

Prove: $REF_{(e_{M1} \parallel e_{N1}) \sqsubseteq (e_{M2} \parallel e_{N2})}$.

Proof. Assume the hypotheses of $REF_{(e_{M1} \parallel e_{N1}) \sqsubseteq (e_{M2} \parallel e_{N2})}$.

$$J_{CM}(v_M, v_N, w_M, w_N) \equiv J_M(v_M, w_M) \wedge J_N(v_N, w_N) \quad \{\text{expanding } J_{CM} \text{ from (17)}\}$$

$$I_M(v_M) \wedge H_M(q_M, w_M) \wedge T_M(q_M, w_M, w'_M) \tag{19}$$

$$I_N(v_N) \wedge H_N(q_N, w_N) \wedge T_N(q_N, w_N, w'_N) \tag{20}$$

Prove: $\exists v'_M, v'_N \cdot G_M(p_M, v_M) \wedge G_N(p_N, v_N) \wedge S_M(p_M, v_M, v'_M) \wedge S_N(p_N, v_N, v'_N) \wedge J_{CM}(v'_M, v'_N, w'_M, w'_N)$. The proof proceeds as follows:

$$\begin{aligned}
 & \exists v'_M, v'_N \cdot G_M(p_M, v_M) \wedge G_N(p_N, v_N) \\
 & \quad \wedge S_M(p_M, v_M, v'_M) \wedge S_N(p_N, v_N, v'_N) \\
 & \quad \wedge J_M(v'_M, w'_M) \wedge J_N(v'_N, w'_N) \quad \{\text{expanding } J_{CM} \text{ from (17)}\} \\
 \equiv & \exists v'_M \cdot G_M(v_M) \wedge S_M(p_M, v_M, v'_M) \wedge J_M(v'_M, w'_M) \\
 & \quad \wedge \exists v'_N \cdot G_N(v_N) \wedge S_N(p_N, v_N, v'_N) \wedge J_N(v'_N, w'_N) \quad \{\text{disjoint } v'_M, v'_N\} \\
 \Leftarrow & REF_{e_{M1} \sqsubseteq e_{M2}} \wedge REF_{e_{N1} \sqsubseteq e_{N2}} \quad \{(16) + (19), (16) + (20)\}
 \end{aligned}$$

We also need to prove the monotonicity for single (non-composed) events that appear at both levels of abstraction. We shall prove it using machines $M1$ and $CM2$. In this case, the gluing invariant described in (17) does not use neither the variables (v_N) neither the invariants (I_N) neither events (e_{N1}) from $N1$. Therefore it can be simplified and rewritten as:

$$J_{CM}(v_M, w_M, w_N) = J_M(v_M, w_M) \wedge J_N(w_N) \tag{21}$$

Deriving from (21), the goal of $INV_{e_{M2} \parallel e_{N2}}$ can be expanded to:

$$j_{CM}(v'_M, w'_M, w'_N) \equiv j_M(v'_M, w'_M) \wedge j_N(w'_N) \tag{22}$$

where j_M and j_N correspond to each invariant from the set of gluing invariants J_M and J_N respectively.

Theorem 6. *An individual event e_{M1} in machine $M1$ is refined by a composed event $e_{M2} \parallel e_{N2}$ in composed machine $CM2$. The monotonicity is preserved if the refinement PO between $M1$ and $M2$ hold in conjunction with the gluing invariant preservation PO for the composed event $e_{M2} \parallel e_{N2}$. The refinement PO between concrete composed event $e_{M2} \parallel e_{N2}$ and abstract non-composed event e_{M1} :*

$$\begin{aligned} REF_{e_{M1} \sqsubseteq (e_{M2} \parallel e_{N2})} : & \quad I_M(v_M) \wedge J_{CM}(v_M, w_M, w_N) \wedge H_M(q_M, w_M) \\ & \quad \wedge H_N(q_N, w_N) \wedge T_M(q_M, w_M, w'_M) \wedge T_N(q_N, w_N, w'_N) \\ & \quad \vdash \exists v'_M \cdot G_M(p_M, v_M) \wedge S_M(p_M, v_M, v'_M) \wedge J_{CM}(v'_M, w'_M, w'_N) \end{aligned} \quad (23)$$

Assume: $REF_{e_{M1} \sqsubseteq e_{M2}}$ and $INV_{e_{M2} \parallel e_{N2}}$.

Prove: $REF_{e_{M1} \sqsubseteq (e_{M2} \parallel e_{N2})}$.

Proof. Assume the hypotheses of $REF_{e_{M1} \sqsubseteq (e_{M2} \parallel e_{N2})}$.

$$\begin{aligned} J_{CM}(v_M, w_M, w_N) & \equiv J_M(v_M, w_M) \wedge J_N(w_N) & \{\text{expanding } J_{CM} \text{ from (21)}\}. \\ I_M(v_M) \wedge H_M(q_M, w_M) \wedge T_M(q_M, w_M, w'_M) & \\ H_N(q_N, w_N) \wedge T_N(q_N, w_N, w'_N) & \end{aligned} \quad (24)$$

And the hypotheses of $INV_{e_{M2} \parallel e_{N2}}$:

$$\begin{aligned} J_{CM}(v_M, w_M, w_N) & \equiv J_M(v_M, w_M) \wedge J_N(w_N) & \{\text{expanding } J_{CM} \text{ from (21)}\} \\ I_M(v_M) \wedge H_M(q_M, w_M) \wedge T_M(q_M, w_M, w'_M) & \\ W_2(v'_M, w_M, w_N, q_M, q_N, w'_M, w'_N) & \end{aligned} \quad (25)$$

$$H_N(q_N, w_N) \wedge T_N(q_N, w_N, w'_N) \quad (26)$$

Prove: $\exists v'_M \cdot G_M(p_M, v_M) \wedge S_M(p_M, v_M, v'_M) \wedge J_{CM}(v'_M, w'_M, w'_N)$. The proof proceeds as follows:

$$\begin{aligned} & \exists v'_M \cdot G_M(p_M, v_M) \wedge S_M(p_M, v_M, v'_M) \\ & \quad \wedge J_{CM}(v'_M, w'_M, w'_N) \\ \equiv & \exists v'_M \cdot G_M(p_M, v_M) \wedge S_M(p_M, v_M, v'_M) \\ & \quad \wedge J_M(v'_M, w'_M) \wedge J_N(w'_N) & \{\text{expanding } J_{CM} \text{ from (21)}\} \\ \equiv & \exists v'_M \cdot G_M(p_M, v_M) \wedge S_M(p_M, v_M, v'_M) \wedge J_M(v'_M, w'_M) \\ & \quad \wedge J_N(w'_N) & \{\text{disjoint } v'_M\} \\ \Leftarrow & REF_{e_{M1} \sqsubseteq e_{M2}} \wedge J_N(w'_N) & \{(16) + (24)\} \\ \Leftarrow & REF_{e_{M1} \sqsubseteq e_{M2}} \wedge INV_{e_{M2} \parallel e_{N2}} & \{(22) + (25) + (26)\} \end{aligned}$$

New events can be added during refinement. They must respect the refinement POs. The refinement PO proof for new events is similar to the previous cases but applied to a single event refined by composed event. Due to the lack of space we do not present it here.

5 Decomposition Guideline

Based on the work developed for composition, its properties and the inverse relation between composition and decomposition, we develop a methodology to partition models in a shared event style. As described in Sect. 3.2, for a shared event decomposition approach, the variables of a system are separated into different sub-components and consequently the rest of the system is decomposed. As a restriction of the shared event approach, no variable sharing is allowed. We present the required steps to process a decomposition, possible problems and how to tackle them.

Variables: From the modeller's point of view, the decomposition starts by defining which sub-components are generated. The following step is to define the partition of variables over the sub-components. The rest of the model decomposition (events, parameters, invariants, contexts) is a consequence of the variable allocation as defined below.

Invariants: The decomposition of the invariants depends on the scope of the variables. Therefore the minimal set of invariants must include the variable type definitions. And these are the required invariants for a valid refinement. Additional ones depend on the user, as they may be useful in later refinements or to help in reusing the sub-components. When an invariant clause is demanded and uses variables placed outside the scope of a sub-component, a further refinement of the composed component might be required to make an explicit separation of the variables.

Events: The partition of the events depends on the partition of the variables. When the decomposition occurs, parameters are shared between the decomposed events. The guard of a decomposed event inherits the guard on the composed event according to the variable partition. For example, let us consider event $e1$:

$$e1 \hat{=} \mathbf{WHEN} \ c = \mathbf{TRUE} \ \mathbf{THEN} \ a := b \ \parallel \ c := \mathbf{FALSE}$$

where variables a and b are of type *DATA* and variable c is a Boolean. This event is enabled when c is **TRUE** and results in a being assigned the value of b and this event being disabled by assigning c to **FALSE**. If this event is decomposed such that variable a belongs to sub-component $M1$ and variables b and c belong to $M2$, then action $a := b$ needs to be split. This assignment needs to be rewritten in a way that these variables are not part of the same expression. A solution is to refine this event in a way that the guards and actions do not refer to variables allocated to different sub-components. Before the decomposition, we refine event $e1$ by adding parameter p :

$$e1 \hat{=} \mathbf{ANY} \ p \ \mathbf{WHEN} \ c = \mathbf{TRUE} \wedge p \in \mathbf{DATA} \wedge p = b \\ \mathbf{THEN} \ a := p \ \parallel \ c := \mathbf{FALSE}$$

Parameter p receives the value of variable b . Then the value of p is assigned to variable a . The parameter p is shared between the sub-components and

whereas variable a is within the scope of $M1$ only containing the guard $p \in DATA$ and the action $a := p$ ($e1'$), the guard $p = b$ is added to $M2$ ($e1''$):

$$e1' \triangleq \mathbf{ANY} \ p \ \mathbf{WHERE} \ p \in DATA \ \mathbf{THEN} \ a := p$$

$$e1'' \triangleq \mathbf{ANY} \ p \ \mathbf{WHERE} \ p \in DATA \wedge p = b \wedge c = \mathbf{TRUE} \ \mathbf{THEN} \ c := \mathbf{FALSE}$$

These corresponds to the value passing of parallel events similar to suggested by Butler [15] for action systems based on CSP: for event $e1''$, parameter p has a output behaviour as it is written by the value of b ; in event $e1'$, parameter p has an input behaviour as its value is read and assigned to variable a .

The events in the sub-components resulting from the decomposition maintain the interface of the original events, preserving the parts corresponding to the variables that belongs to each sub-component.

6 File Access Management Case Study

A distributed system is presented where a system is decomposed into two smaller parts. A specification of a file management system is developed: files containing *DATA* can be created, read, overwritten, deleted and sent to other users. Each file has an owner. The owners are users with clearance level ranging from 1 to 10 where 10 is the highest level. A *super* user exists with clearance level 10. Moreover, files have a classification level varying from 1 to 10. Permission is needed in order to read, modify or delete a file. When the permission is granted, the requested action can take place.

Machine *FileAccessManagement* contains variables *user*, *file*, *fileData* (contains the data of each file) and *fileStatus* (defines the status of a file operation and can have the states *SUCCESS* or *FAILED*). When a file is created or sent, variable *fileStatus* is updated accordingly to the result of the operation. The status of a file must be reset (in event *clearFileStatus*) to allow a new operation in the same file. The access management is defined by variables *userClearanceLevel*, *permission*, *fileClassification* and *fileOwner*. A user can change the clearance of another user as long as the former has a clearance level superior to the latter as described in event *modifyUser* (guard *grd3* in Fig. 6). For all the other operations, permission is required and it is granted by the non-deterministic action in event *requestPermission*. When a permission is granted, a file can be read, modified, deleted or sent to another user. A file can only be modified by users with a clearance level superior to the file classification (guard *grd8* in event *overwriteFile*). To delete a file, described in event *deleteFile*, the user must be the owner of the file or be the *super user* as described by guard *grd5*.

Our intention is to separate the management of permissions (administrative task) from the modification of the files in the disk (writing, reading tasks). The result are two sub-components, *AccessMng* and *FileMng* that deal with different

```

variables userClearanceLevel permission
           fileClassification fileOwner user file
           fileData fileStatus

invariants
@inv1 file ∈ FILE
@inv2 user ∈ USER
@inv3 userClearanceLevel ∈ user → ClearanceLevel
@inv4 permission ∈ PERMISSION
@inv5 fileClassification ∈ file → Classification
@inv6 fileOwner ∈ file → user
@inv7 fileData ∈ file → DATA
@inv8 fileStatus ∈ file → STATUS
@inv9 ran(fileStatus) ⊆ {SUCCESS, FAILED}
@inv10 fileOwner ∈ file → user
@inv11 ∀f.f ∈ file ⇒
    userClearanceLevel(fileOwner(f)) > fileClassification(f)

event addUser
any uu //changed user
    masterUser //user ordering the change
    newUserClearanceLevel //new ClearanceLevel
where
@grd1 uu ∈ dom(userClearanceLevel)
@grd2 newUserClearanceLevel ∈ ClearanceLevel
@grd3 newUserClearanceLevel
    < userClearanceLevel(uu)
@grd4 masterUser ≠ uu
@grd5 uu ≠ super
@grd6 ∀f.f ∈ dom(fileClassification)
    ∧ fileOwner(f)=uu ⇒
    newUserClearanceLevel>fileClassification(f)
@grd7 uu ≠ user
@grd8 masterUser ∈ user
then
@act1 userClearanceLevel(uu)=
    newUserClearanceLevel
@act2 user = user ∪ {uu}
end

event sendFile
any ff recipient u fs cl
where
@grd1 ff ∈ file
@grd2 u ∈ user
@grd3 recipient ∈ user
@grd4 ff ∈ dom(fileStatus)
@grd5 fs ∈ {SUCCESS, FAILED}
@grd6 u ≠ recipient
@grd7 uedom(userClearanceLevel)
@grd8 cl ∈ Classification
@grd9 permission = ALLOWED
@grd10 ff ∈ dom(fileClassification)
    ⇒ cl = fileClassification(ff)
@grd11 userClearanceLevel(u)>cl
then
@act1 fileStatus(ff) = fs
@act2 fileClassification(ff) = cl
@act3 permission = OFF
@act4 fileOwner(ff) = u
end

event requestPermission
where
@grd1 permission ≠ ALLOWED
then
@act1 permission := PERMISSION \ {OFF}
end

event clearFileStatus
any ff
where
@grd1 ff ∈ dom(fileStatus)
@grd2 fileStatus(ff)
    ∈ {SUCCESS, FAILED}
then
@act1 fileStatus = {ff} ← fileStatus
end

event modifyUser
any uu // changed user
    masterUser // user ordering the change
    newUserClearanceLevel //new ClearanceLevel
where
@grd1 uu ∈ dom(userClearanceLevel)
@grd2 newUserClearanceLevel ∈ ClearanceLevel
@grd3 newUserClearanceLevel
    < userClearanceLevel(uu)
@grd4 masterUser ≠ uu
@grd5 uu ≠ super
@grd6 ∀f.f ∈ dom(fileClassification)
    ∧ fileOwner(f)=uu ⇒
    newUserClearanceLevel>fileClassification(f)
then
@act1 userClearanceLevel(uu)=
    newUserClearanceLevel
end

event deleteFile
any ff //file to be deleted
    u //user executes action
where
@grd1 ff ∈ file
@grd2 u ∈ user
@grd3 permission = ALLOWED
@grd4 ff ∈ dom(fileOwner)
@grd5 u ∈ {super, fileOwner(ff)}
then
@act1 file=file \ {ff}
@act2 fileData={ff} ← fileData
@act3 fileStatus={ff} ← fileStatus
@act4 fileClassification=
    {ff} ← fileClassification
@act5 permission = OFF
@act6 fileOwner={ff} ← fileOwner
end

event overwriteFile
any ff dd cl u
where
@grd1 ff ∈ file
@grd2 dd ∈ DATA
@grd3 dd ≠ fileData(ff)
@grd4 uedom(userClearanceLevel)
@grd5 cl ∈ Classification
@grd6 permission = ALLOWED
@grd7 ff ∈ dom(fileClassification)
    ⇒ cl = fileClassification(ff)
@grd8 userClearanceLevel(u)>cl
then
@act1 fileData(ff)=dd
@act2 fileClassification(ff)= cl
@act3 permission = OFF
@act4 fileOwner(ff) = u
end

```

Fig. 6. *FileAccessManagement*: variables, invariants and some events

aspects of the system. An advantage of this separation is to more easily define specific properties to each part without additional constraints of the other part. For instance, an administrative task of *AccessManagement* is to have a quota of disk per user which is irrelevant to *FileMng*. Overwriting a file in the disk is relevant to *FileMng* but not to *AccessMng* that deals with the users that are allowed to execute this action. Therefore we decompose *FileAccessManagement* into two sub-components as described in the next section.

6.1 Decomposition: *AccessMng* and *FileMng*

Following the steps suggested in Sect. 5, the variables of *FileAccessManagement* are allocated to *AccessMng* and *FileMng* as described in the following table:

	FileMng	AccessMng
Variables	file,user, fileData,fileStatus	userClearanceLevel,permission, fileOwner,fileClassification

The distribution of events can be seen on the composed machine described in Fig. 7. Some events are specific to a sub-component: events *modifyUser* and *requestPermission* belong to *AccessMng* while *clearFileStatus* belongs to *FileMng*; the other events are shared. In Fig. 8, the invariants include theorems defining the variable types as suggested in Sect. 5. Moreover for Fig. 8(b), invariants relating variables for the same sub-component are automatically included. Figure 9 shows the decomposed events *overwriteFile* where parameters *ff*, *dd* and *cl* are shared (value passing from *AccessMng* to *FileMng*). Also the actions are split according to the user's variable selecting (cf. Table above): *fileOwner*, *fileClassification* and *permission* belong to *AccessMng* while *fileData* belongs to *FileMng*.

```

COMPOSED MACHINE FileAccessManagement
INCLUDES
  AccessMng, FileMng
EVENTS
  addUser  $\hat{=}$  AccessMng.addUser || FileMng.addUser
  modifyUser  $\hat{=}$  AccessMng.modifyUser
  createFile  $\hat{=}$  AccessMng.createFile || FileMng.createFile
  readFile  $\hat{=}$  AccessMng.readFile || FileMng.readFile
  overwriteFile  $\hat{=}$  AccessMng.overwriteFile || FileMng.overwriteFile
  deleteFile  $\hat{=}$  AccessMng.deleteFile || FileMng.deleteFile
  sendFile  $\hat{=}$  AccessMng.sendFile || FileMng.sendFile
  requestPermission  $\hat{=}$  AccessMng.requestPermission
  clearFileStatus  $\hat{=}$  FileMng.clearFileStatus

```

Fig. 7. Composed machine *FileAccessManagement*

<pre> machine <i>AccessMng sees</i> User_C0 <i>AccessManagement_C0 FileManagement_C0</i> variables userClearanceLevel permission fileClassification fileOwner invariants theorem @typing_userClearanceLevel userClearanceLevel \in P(USER \times Z) theorem @typing_fileOwner fileOwner \in P(FILE \times USER) theorem @typing_permission permission \in PERMISSION theorem @typing_fileClassification fileClassification \in P(FILE \times Z) </pre>	<pre> machine <i>FileMng sees</i> User_C0 <i>AccessManagement_C0 FileManagement_C0</i> variables file user fileData fileStatus invariants theorem @typing_fileStatus fileStatus \in P(FILE\timesSTATUS) theorem @typing_file file \in P(FILE) theorem @typing_user user \in P(USER) theorem @typing_fileData fileData \in P(FILE \times DATA) @FileAccessMng_inv1 file \in FILE @FileAccessMng_inv2 user \in USER @FileAccessMng_inv7 fileData \in file \rightarrow DATA @FileAccessMng_inv8 fileStatus \in file \leftrightarrow STATUS @FileAccessMng_inv9 ran(fileStatus) \subset {SUCCESS, FAILED} </pre>
(a)	(b)

Fig. 8. *AccessMng* (a) and *FileMng* (b): variables and invariants

Composition and decomposition are combined: the decomposition partitions the model in sub-components based on the variables and the composition expresses the events' interaction. The extensibility of Rodin, allows new functionalities to be added to the Event-B language. *Silva et al* [19] developed a semi-automatic decomposition tool for shared event or shared variable. A composition tool [20] is also available in the Rodin platform. We use both tools: *FileAccessManagement* is decomposed using the decomposition tool and the composition tool shows

the event splitting. In a shared event decomposition, the user does not control the event splitting since they are a consequence of the variable allocation (selected by the user). Therefore the composition view gives an additional insight of the entire process, complementing the decomposition view.

As we proved in Sect. 4.3, shared event composition is monotonic and consequently sub-components can be further refined independently preserving the verified properties while composed. For instance, *AccessMng* can be refined by defining a more deterministic event *requestPermission* based on the kind of operation and user. For *FileMng*, event *sendFile* can be further refined by introducing a processing queue where events can be stored. The advance of independent refinement of sub-components is a separation of behaviours and properties verifiable without the interference of other sub-components.

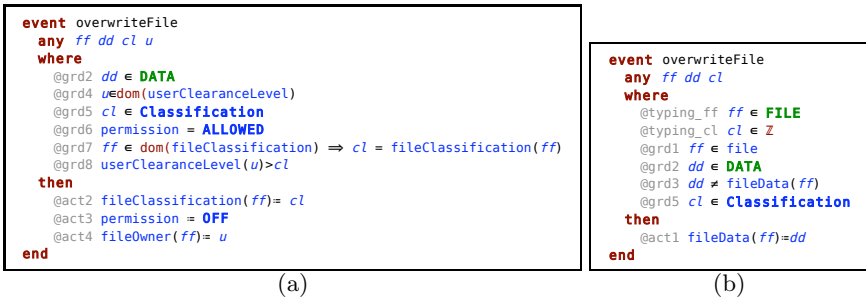


Fig. 9. Decomposed events *overwriteFile* for *AccessMng* (a) and *FileMng* (b)

7 Related Work

Composition allows the interaction of sub-components. Back [21], Abadi and Lamport [22] studied the interaction of components through shared variable composition. Jones [23] also proposes a shared variable composition for VDM by restricting the behaviour of the environment and the operation itself in order to consider the composition valid using rely-guarantee conditions. In Z, composition can be achieved by combining schemas [14] where variables within the same scope cannot have identical names or by views [1] allowing the development of partial specifications that can interact through invariants that relate their state or by operations' synchronisation. Although systems are developed in single machines in classical B, Bellegarde et al [24] suggest a composition by rearranging separated machines and synchronising their operations under feasibility conditions. The behaviour of a component composition is seen as a labelled transition system using weakest preconditions, where a set of authorised transitions are defined. The objective is to verify the refinement of synchronised parallel composition between components but it is limited to finite state transitions and a finite number of components. This work differs from ours as it uses a labelled transition system including a notion of refinement and variable sharing while we use synchronisation and communication in the CSP style. Butler and Walden [25]

discuss a combination of action systems and classical B by composing machines using parallel systems in an action system style and preserving the invariants of the individual machines. This approach allows the classical B to derive parallel and distributed systems and since the parallel composition of action system is monotonic, the sub-systems in a parallel composition may be refined independently. This work is closely related to our work with similar underlying semantics and notion of refinement based on CSP. Abrial et al [6] propose a state-based decomposition for Event-B introducing the notion of shared variables and external events. Although it allows variable sharing, this approach is also monotonic but its respective nature is more suitable for parallel programs [26].

8 Conclusions

Our Event-B composition and decomposition is based on the close relation between action systems and Event-B plus the correspondence between action systems and CSP as described in Sect. 3.1. Composition POs are defined to ensure valid composed machines and refinements. These can be simplified when machine POs are reused. We prove that shared event composition is monotonic by means of POs and “top-down” refinement is allowed. Sub-components interact through event parameters by value-passing. Event-B is extended to support *shared event composition*, allowing combination and reuse of existing sub-components through the introduction of *composed machines*. We do not address the step corresponding to the translation of the composition to an implementation. This study needs to be carried out in the future. Using a case study, composition, decomposition and refinement are combined, suggesting a methodology for modelling distributed systems and verifying properties through the generation of POs. A file access management system is decomposed into two independent parts with a separation of their logics: file and access management and possible refinements are suggested. Other case studies have been applying (de)composition with success such as the decomposition of a safety metro system [2].

References

1. Jackson, D.: Structuring Z specifications with views. *ACM Trans. Softw. Eng. Methodol.* 4(4), 365–389 (1995)
2. Zave, P., Jackson, M.: Conjunction as Composition. *ACM Trans. Softw. Eng. Methodol.* 2(4), 379–411 (1993)
3. Jones, C.B.: Wanted: a compositional approach to concurrency. In: *Programming Methodology*, pp. 5–15. Springer-Verlag New York, Inc, New York (2003)
4. Poppleton, M.: The Composition of Event-B Models. In: Börger, E., Butler, M., Bowen, J.P., Boca, P. (eds.) *ABZ 2008. LNCS*, vol. 5238, pp. 209–222. Springer, Heidelberg (2008)
5. Abrial, J.R.: *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, Cambridge (2010)
6. Abrial, J.R., Hallerstede, S.: Refinement, Decomposition, and Instantiation of Discrete Models: Application to Event-B. *Fundam. Inf.* 77(1-2), 1–28 (2007)

² This case study is available online at <http://eprints.ecs.soton.ac.uk/22195/>

7. Butler, M.: An Approach to the Design of Distributed Systems with B AMN. In: Till, D., P. Bowen, J., Hinchey, M.G. (eds.) ZUM 1997. LNCS, vol. 1212, pp. 221–241. Springer, Heidelberg (1997)
8. Hoare, C.A.R.: Communicating Sequential Processes. Prentice Hall International Series in Computer Science (1985)
9. Morgan, C.: Of wp and CSP. In: Beauty is our Business: a Birthday Salute to Edsger W. Dijkstra, pp. 319–326. Springer-Verlag New York, Inc., New York (1990)
10. Back, R.-J.R., Kurki-Suonio, R.: Decentralization of Process Nets with Centralized Control. In: PODC 1983: Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing, pp. 131–142. ACM, New York (1983)
11. Abrial, J.R.: The B-Book: Assigning programs to meanings. Cambridge University Press, Cambridge (1996)
12. Rodin: RODIN project Homepage (September 2008), <http://rodin.cs.ncl.ac.uk> (accessed July 27, 2010)
13. Abrial, J.R., Butler, M.J., Hallerstede, S., Voisin, L.: An Open Extensible Tool Environment for Event-B. In: Liu, Z., Kleinberg, R.D. (eds.) ICFEM 2006. LNCS, vol. 4260, pp. 588–605. Springer, Heidelberg (2006)
14. Spivey, J.M.: The Z Notation: a Reference Manual. Prentice-Hall, Inc., Englewood Cliffs (1989)
15. Butler, M.J.: A CSP Approach to Action Systems. PhD thesis, Oxford University (1992)
16. Butler, M.: Stepwise Refinement of Communicating Systems. *Science of Computer Programming* 27(2), 139–173 (1996)
17. Butler, M.: Synchronisation-Based Decomposition for Event-B. In: RODIN Deliverable D19 Intermediate Report on Methodology, pp. 47–57 (2006)
18. Abrial, J.R., Butler, M., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: An Open Toolset for Modelling and Reasoning in Event-B. *International Journal on Software Tools for Technology Transfer, STTT* (April 2010)
19. Silva, R., Pascal, C., Hoang, T.S., Butler, M.: Decomposition Tool for Event-B. *Software: Practice and Experience* 41(2), 199–208 (2011)
20. Silva, R., Butler, M.: Parallel Composition Using Event-B (July 2009), http://wiki.event-b.org/index.php/Parallel_Composition_using_Event-B (accessed July 27, 2010)
21. Back, R.-J.R.: Refinement Calculus, part II: Parallel and Reactive Programs. In: de Bakker, J.W., de Roever, W.-P., Rozenberg, G. (eds.) REX 1989. LNCS, vol. 430, pp. 67–93. Springer, Heidelberg (1990)
22. Abadi, M., Lamport, L.: Composing Specifications. In: de Bakker, J.W., de Roever, W.P., Rozenberg, G. (eds.) REX 1989. LNCS, vol. 430, pp. 1–41. Springer, Heidelberg (1990)
23. Woodcock, J., Dickinson, B.: Using VDM with Rely and Guarantee-Conditions. In: Bloomfield, R.E., Jones, R.B., Marshall, L.S. (eds.) VDM 1988. LNCS, vol. 328, pp. 434–458. Springer, Heidelberg (1988)
24. Bellegarde, F., Julliand, J., Kouchnarenko, O.: Synchronized Parallel Composition of Event Systems in B. In: Bert, D., Bowen, J.P., Henson, M.C., Robinson, K. (eds.) B 2002 and ZB 2002. LNCS, vol. 2272, pp. 436–457. Springer, Heidelberg (2002)
25. Butler, M., Waldén, M.: Distributed System Development in B. Technical Report TUCS-TR-53, Turku Centre for Computer Science, 14 (1996)
26. Hoang, T., Abrial, J.R.: Event-B Decomposition for Parallel Programs. In: Frappier, M., Glässer, U., Khurshid, S., Laleau, R., Reeves, S. (eds.) ABZ 2010. LNCS, vol. 5977, pp. 319–333. Springer, Heidelberg (2010)

ABS: A Core Language for Abstract Behavioral Specification^{*}

Einar Broch Johnsen¹, Reiner Hähnle², Jan Schäfer³,
Rudolf Schlatte¹, and Martin Steffen¹

¹ Department of Informatics, University of Oslo, Norway
{einarj,rudi,msteffen}@ifi.uio.no

² Chalmers University of Technology, Sweden
reiner@chalmers.se

³ Department of Computer Science, University of Kaiserslautern
jschaefer@cs.uni-kl.de

Abstract. This paper presents ABS, an *abstract behavioral specification* language for designing executable models of distributed object-oriented systems. The language combines advanced concurrency and synchronization mechanisms for concurrent object groups with a functional language for modeling data. ABS uses asynchronous method calls, interfaces for encapsulation, and cooperative scheduling of method activations inside concurrent objects. This feature combination results in a concurrent object-oriented model which is inherently compositional. We discuss central design issues for ABS and formalize the type system and semantics of Core ABS, a calculus with the main features of ABS. For Core ABS, we prove a subject reduction property which shows that well-typedness is preserved during execution; in particular, “method not understood” errors do not occur at runtime for well-typed ABS models. Finally, we briefly discuss the tool support developed for ABS.

1 Introduction

This paper presents ABS, an *abstract behavioral specification* language for distributed object-oriented systems. Abstract behavioral specification languages can be situated between design-oriented and implementation-oriented specification languages. ABS addresses the specification of *executable formal models* for distributed object-oriented systems: it allows a high-level specification of a system, including its concurrency and synchronization mechanisms as well as local state updates. Thus ABS models capture the concurrent control flow of object-oriented systems, yet abstract away from many implementation details which may be undesirable at the modeling level, such as the concrete representation of internal data structures, the scheduling of method activations, and the properties of the communication environment.

^{*} Partly funded by the EU project FP7-231620 HATS: Highly Adaptable and Trustworthy Software using Formal Models (<http://www.hats-project.eu>).

The target domain of ABS is *distributed systems*. In a distributed setting, the implementation details of other objects in the system are not necessarily known. Instead, ABS uses interfaces as types for objects, abstracting in the type system from the classes implementing the functionality of these objects. The strict separation of types and implementations makes concurrent ABS models *compositional*. The concurrency model of ABS is similar to that of JCoBox [34], which generalizes the concurrency model of Creol [24] from single concurrent objects to concurrent object groups.

The language supports asynchronous method calls, which trigger activities in other objects without transferring control from the caller, using first-class futures [13]. Thus, an object may have many method activations competing to be executed. A distinguishing feature of this concurrency model is the use of *cooperative scheduling* of method activations to explicitly control the internal interleaving of activities inside concurrent object groups. Thus, a clear notion of quiescent state may be formulated, namely when the active process of each object in the cog is *idle*. This allows an approach to system verification in which local reasoning is based on the maintenance of monitor invariants which must hold in quiescent states. Because of cooperative scheduling and the interface encapsulation mechanism, local reasoning about the concurrent object system can be done by suitable extensions of standard verification systems for sequential object-oriented programs. This approach is carried out in a number of papers [3, 13, 15, 18] for both Creol and Core ABS.

The present paper discusses the design decisions behind ABS and defines Core ABS, a calculus formalizing the main features of ABS. The contributions of this paper may be summarized as follows:

- We define the *functional level* of ABS, which is used to abstract computations on internal data in concurrent objects. ABS supports user-defined parametric data types and functions with pattern matching. We define a syntax, type system, and reduction system for functional expressions in Core ABS.
- We define the *concurrent object level* of ABS, which is used to capture concurrent control flow and communication in ABS models. This part of ABS integrates functional expressions, imperative object-based programming, and concurrent object groups with cooperative scheduling. We define a syntax, type system, and an SOS style operational semantics for the concurrent object level of Core ABS.
- We show how *type preservation* is guaranteed at runtime for well-typed models in Core ABS, with a particular focus on the creation of concurrent object groups, objects, and first-class futures.

An extended discussion and further technical details on the ABS language may be found in [14] while a program logic for Core ABS is in [15, Chap. 2].

2 Abstract Behavioral Specification

Specification languages can be categorized into three categories with partly complementary and partly overlapping purposes:

- *Design-oriented languages* focus on structural aspects of systems, such as the relationship between features or classes, and the flow of messages between objects. Examples of design-oriented languages are UML/OCL [36], FDL [35], and architectural description languages [11, 29].
- *Foundational languages* focus on foundational aspects of, e.g., concurrency and interaction, by identifying a small set of primitives and their formal semantics. Examples of foundational languages are process algebras [31], automata models [27], and object calculi [1, 23].
- *Implementation-oriented languages* focus on behavioral properties of implemented systems. Examples of implementation-oriented specification languages are JML [7] and Spec# [6].

Design-oriented languages often provide visual means of displaying a system's structure, but typically lack flexible constructs for expressing concurrency and synchronization aspects of a system. Foundational languages address this concern, but their minimalistic set of language features makes it cumbersome to develop models of real systems without complicated encodings; the resulting models typically do not reflect the structure of an object-oriented target program. Even the abstractions of object calculi make it difficult to express real systems; for example, Featherweight Java [23] does not provide fields in objects. In contrast, implementation-oriented specification languages are restricted to the particular, often extremely complex, concurrency and synchronization mechanisms of their target language, and typically enforce particular solutions which may be undesirable at the design stage.

ABS is situated between these three categories of specification languages. It has in common with implementation-oriented languages that it is designed to be close to the way programmers think, by maintaining a Java-like syntax and a control flow close to an actual implementation. In fact, ABS models may be automatically compiled into, e.g., Java (see Sect. 7). On the other hand, the language has a formally defined semantics, in the style of foundational languages, and allows the modeler to abstract from undesirable implementation details by means of user-defined algebraic data types and functions. Consequently, imperative structures may be used to study particular aspects of a system, while other aspects may be abstracted to ADTs. In addition, the concurrency model of ABS abstracts from particular assumptions about the communication environment, such as ordering schemes for message transfer and scheduling policies for the selection of method activations inside objects.

3 The Design of ABS

3.1 The Overall Structure of ABS

ABS targets distributed object-oriented systems. The concurrency model of ABS is *two-tiered* and separates local, synchronous, shared-memory communication in the *lower tier* from asynchronous communication with only message passing in the *upper tier*. The lower tier is inspired by JCoBox [34] which generalizes

the concurrency model of Creol [13, 24] from concurrent objects to concurrent object groups, so-called *cogs*. Cogs can be seen as object-based runtime components with their own object heaps. A cog's behavior is based on cooperative multi-tasking of external requests and synchronous internal method activations. Cooperative multi-tasking guarantees data-race freedom inside a cog and enables active and reactive behavior to be safely combined. Only asynchronous method calls can occur between different cogs, different cogs have no shared heap.

Complementing the concurrent object language, ABS supports user-defined data types with (first-order) functions and pattern matching. This functional level of ABS is largely orthogonal to the concurrent object level and is intended to model data manipulation. Such data is immutable and can safely be exchanged between cogs. Using functional data types to realize most internal data structures in the cogs can significantly simplify the specification and verification of models. The value of functional expressions can be underspecified which is important in order to realize abstraction from concrete implementations.

ABS contains non-deterministic constructs, notably, the outcome of executing concurrency primitives is non-deterministic. While underspecification is used for data abstraction, non-deterministic execution semantics is the prerequisite for abstracting behavior. As a modeling language, ABS makes no a priori assumptions about, e.g., concrete scheduling mechanisms. Importantly, underspecification and non-determinism do not preclude executability: the outcome of a non-deterministic transition step is a *finite set* of possible successor states which can be systematically inspected in simulation, analysis, and visualization tools.

In the remainder of this section, we briefly describe how to represent and work with data, and then discuss the concurrent object level of ABS.

3.2 Data Types, Functions, and Pattern Matching

ABS does *not* have primitive types for basic values. Instead, algebraic data types may be defined by the user. A library of predefined data types and operators is provided, including `Unit`, `Bool`, `Int`, and `String`. Data types and functions in ABS can be *polymorphic*; i.e., their definition may have type parameters.

Example 1. The following code shows the polymorphic data type `List<A>` (part of the ABS Standard Library), as well as a function `contains` which checks whether an element `e` is a member of a given list `l`.

```
data List<A> = Nil | Cons(A, List<A>);
def Bool contains<A>(List<A> l, A e) =
  case l { Nil => False;
          Cons(e, _) => True;
          Cons(_, xl) => contains(xl, e); };
```

3.3 Interfaces in ABS

ABS is a class-based language, which uses interfaces for typing. ABS has no class inheritance, but multiple inheritance is allowed at the interface level. A

class may implement several interfaces, provided that it supports all methods offered by these interfaces. Subtype polymorphism is permitted at the level of interfaces: *an object supporting an interface I may be replaced by another object supporting I or a subtype of I* in a context where *I* is expected, although the classes of the two objects may differ.

Due to the typing of object variables by interfaces, the fields of another object cannot be accessed directly, only method calls to the object are possible. The object controls its own state; another object can only manipulate the state indirectly via the methods exported through an interface. In fact, interfaces are the only encapsulation mechanism of ABS objects and no access modifiers are provided. Since the class may support several interfaces, different methods may be exported to the environment through different interfaces; for example, a *super user* interface may export methods not seen through the normal *user* interface.

Example 2. Consider a peer-to-peer system whose participant *nodes* act both as *servers* and *clients*, and exchange *files* which are composed of *packets*. In this setting, it is important for each node to remain responsive during file transfer, both to serve concurrent requests and for simultaneous downloads. Files are transferred packet by packet, with one request per packet.

Client behavior is modeled by a `Client` interface, declaring a `getFile` method which is invoked from the outside, e.g., via a graphical user interface. Server behavior consists of a method `getFilenames` for querying the server about its available files, a method `getLength` for querying the length in packets for a given file, and a method `getPack` which requests the *n*'th packet of a given file.

```

type Packet = String;
type File = List<Packet>;

interface Server {
  List<Filename> getFilenames();
  Int getLength(Filename fId);
  Packet getPack(Filename fId,
    Int pNbr);
}

interface Client {
  File getFile(Server sId,
    Filename fId);
}

interface Peer
extends Client, Server {
}

```

3.4 The Concurrency Model of ABS

Conceptually, each cog has a dedicated processor and lives in a distributed environment with asynchronous and unordered communication. A set of objects is located within a cog. On the upper tier of the ABS concurrency model, all communication is between named objects, typed by interfaces, by means of asynchronous method calls. Calls are asynchronous as the caller may decide at runtime when to synchronize with the reply from a call. Asynchronous method calls may be seen as triggers of concurrent activity, spawning new method activations (so-called *processes*) in the called object.

Active behavior, triggered by an optional method *run*, is interleaved with passive behavior, triggered by asynchronous method calls. Thus, an object has

a set of processes to be executed, which originate from method activations. Among these, at most one process among the objects of a cog is *active* and the other processes are *suspended* in a process pool. Process scheduling is non-deterministic, but controlled by *processor release points* in a cooperative way. Hence, the amount of concurrency in an ABS model is directly reflected in the number of cogs introduced in the model. A Creol-like concurrent object model corresponds to an ABS model in which each object has its own cog.

Example 3. Consider a class `Node` which implements the peer behavior of Example 2, providing both client and server functionality. The `getFile` method first obtains the length of the requested file, then fetches the packets one by one. As it uses asynchronous method calls, the object can interleave the execution of `getFile` with answering requests in its server role and other invocations of `getFile`. (For brevity the implementation of the fields and remaining methods, such as `getFilenames`, is omitted. For a full model in ABS, see [14, App. E].)

```

class Node implements Peer {
    // Fields and other methods
    // of Node omitted

    File getFile(Server sId,
                 Filename fId) {
        File file = Nil;
        Fut<Int> l1 = sId!getLength(fId); return file;
        await l1?;
        Int lth = l1.get;
    }

    while (lth > 0) {
        lth = lth - 1;
        Fut<Packet> l2
            = sId!getPack(fId, lth);
        await l2?;
        Packet pack = l2.get;
        file = Cons(pack, file);
    }
}

```

4 A Formal ABS Calculus

Core ABS is a formal calculus which simplifies the full ABS language by excluding, e.g., the module system, type synonyms, the predefined data types (except `Bool`), and annotations. However, Core ABS captures the central features of ABS. (A complete formalization of ABS exists in the rewriting logic of Maude [10].)

4.1 The Syntax of Core ABS

An ABS *model* P defines interfaces, classes, datatypes, functions, and a *main block* to configure the initial state (see Fig. 2). Objects are dynamically created from classes with attributes initialized to type-correct default values (e.g., `null` for object references) that may be reassigned in an optional method *init*.

A Functional Language for User-Defined Parametric Data Types and Functions. The functional level of Core ABS defines data types and functions, as shown in Fig. 1. The ground types T consist of basic types B such as `Bool` and `Int`, as well as names D for data types and I for interfaces. In general, a type A

<i>Syntactic categories</i>	<i>Definitions</i>
T in Ground Type	$T ::= B \mid I \mid D \mid D(\overline{T})$
B in Basic Type	$B ::= \text{Bool} \mid \text{Int} \mid \dots$
A in Type	$A ::= N \mid T \mid D(\overline{A})$
N in Names	$Dd ::= \text{data } D[\overline{A}] = \text{Cons}[\overline{Cons}];$
x in Variable	$\text{Cons} ::= \text{Co}[\overline{A}]$
e in Expression	$F ::= \text{def } A \text{ fn}[\overline{A}](\overline{A} \overline{x}) = e;$
b in Bool Expression	$e ::= b \mid x \mid t \mid \text{this} \mid \text{destiny} \mid \text{Co}[\overline{e}] \mid \text{fn}(\overline{e}) \mid \text{case } e \{ \overline{br} \}$
t in Ground Term	$t ::= \text{Co}[\overline{t}] \mid \text{null}$
br in Branch	$br ::= p \Rightarrow e;$
p in Pattern	$p ::= - \mid x \mid t \mid \text{Co}[\overline{p}]$

Fig. 1. Core ABS syntax for the functional level. Terms \overline{e} and \overline{x} denote possibly empty lists over corresponding syntactic categories, and square brackets $[\]$ optional elements.

may also contain type variables N (i.e., uninterpreted type names [32]). In data type declarations Dd , a data type D has at least one constructor Cons , which has a name Co and a list of types \overline{A} for its arguments. Function declarations F consist of a return type A , a function name fn , a list of variable declarations \overline{x} of types \overline{A} , and an expression e . Data type declarations Dd and function declarations F may optionally have type parameters. *Expressions* e include Boolean expressions b , variables x , (ground) terms t , the self-identifier **this**, the return address **destiny** of the method activation, constructor expressions $\text{Co}(\overline{e})$, function expressions $\text{fn}(\overline{e})$, and case expressions **case** $e \{ \overline{br} \}$. Ground terms t are constructors applied to ground terms $\text{Co}(\overline{t})$, and **null**. Case expressions have a list of branches $p \Rightarrow e$, where p is a pattern. The branches of case expressions are evaluated in the listed order. Patterns include wild cards $-$, variables x , terms t , and constructor patterns $\text{Co}(\overline{p})$. Let the function $\text{vars}(p)$ return the set of *variables* in a pattern p , defined inductively by $\text{vars}(-) = \text{vars}(t) = \emptyset$, $\text{vars}(x) = \{x\}$, and $\text{vars}(\text{Co}(p_1, \dots, p_n)) = \bigcup_{i=1}^n \text{vars}(p_i)$.

The Concurrent Object Level of Core ABS is given in Fig. 2. An interface IF has a name I and method signatures Sg . A class CL has a name C , interfaces \overline{T} (specifying types for its instances), formal parameters and state variables \overline{x} of types \overline{T} , and methods \overline{M} . (The *fields* of the class are both its parameters and state variables). A method signature Sg declares the return type T of a method with name m and formal parameters \overline{x} of types \overline{T} . M defines a method with signature Sg , local variable declarations \overline{x} of types \overline{T} , and a statement s . Statements may access attributes of the current class, locally defined variables, and the method's formal parameters. A program's main block is a method body $\{ \overline{T} \overline{x}; s \}$. There are no type variables at the concurrent object level of ABS.

Right-hand side expressions rhs include object creation within the same cog (written “**new** $C(\overline{e})$ ”) and in a fresh cog (written “**new cog** $C(\overline{e})$ ”), method calls, and (pure) expressions e . Statements are standard for sequential composition, assignment, **skip**, **if**, **while**, and **return** constructs. The statement **suspend** unconditionally releases the processor, suspending the active process. In **await** g , the guard g controls processor release and consists of Boolean

<i>Syntactic categories</i>	<i>Definitions</i>
C, I, m in Names	$P ::= \overline{Dd} \overline{F} \overline{IF} \overline{CL} \{\overline{T} \overline{x}; s\}$
g in Guard	$IF ::= \mathbf{interface} I \{\overline{Sg}\}$
s in Statement	$CL ::= \mathbf{class} C [(\overline{T} \overline{x})] [\mathbf{implements} \overline{I}] \{\overline{T} \overline{x}; \overline{M}\}$
	$Sg ::= T m (\overline{T} \overline{x})$
	$M ::= Sg \{\overline{T} \overline{x}; s\}$
	$g ::= b \mid e? \mid g \wedge g$
	$s ::= s; s \mid x = rhs \mid \mathbf{suspend} \mid \mathbf{await} g \mid \mathbf{skip}$
	$\quad \mid \mathbf{if} b \{s\} [\mathbf{else} \{s\}] \mid \mathbf{while} b \{s\} \mid \mathbf{return} e$
	$rhs ::= e \mid \mathbf{new} [\mathbf{cog}] C[(\overline{e})] \mid e!m(\overline{e}) \mid e.m(\overline{e}) \mid x.\mathbf{get}$

Fig. 2. Core ABS syntax for the concurrent object level

conditions b and return tests $x?$ (see below). If g evaluates to false, the processor is released and the process *suspended*. When the processor is idle, any enabled process from the object's pool of suspended processes may be scheduled. Consequently, explicit signaling is redundant in ABS.

Communication in ABS is based on asynchronous method calls, denoted $o!m(\overline{e})$, and synchronous method calls, denoted $o.m(\overline{e})$, where o is an object expression (i.e., an expression typed by an interface). Any method may be called either synchronously or asynchronously. After asynchronously calling $x = o!m(\overline{e})$, the caller may proceed with its execution without blocking on the call. Here x is a future variable; i.e., a variable which refers to a return value which has yet to be computed. There are two operations on future variables, which explicitly control external synchronization in ABS. Let e be an expression denoting a future variable. First, a return test $e?$ evaluates to false unless the reply to the call can be retrieved. (Return tests are used in guards.) Second, the return value is retrieved by the expression $e.\mathbf{get}$, which blocks all execution in the object until the return value is available.

When executed between objects in *different* cogs, then the statement sequence $x = o!m(\overline{e}); v = x.\mathbf{get}$ amounts to a blocking, *synchronous call* and is abbreviated $v = o.m(\overline{e})$. In contrast, synchronous calls $v = o.m(\overline{e})$ *inside* a cog have the reentrant semantics known from, e.g., Java threads. The statement sequence $x = o!m(\overline{e}); \mathbf{await} x?; v = x.\mathbf{get}$ codes a non-blocking, *preemptable call*, abbreviated $\mathbf{await} v = o.m(\overline{e})$. In many cases, these method calls with *implicit* futures provide sufficiently flexible concurrency control to the modeler.

4.2 The Type System of Core ABS

A *mapping* binds names to values. Let Γ be a mapping, $[N \mapsto V]$ a binding from name N to value V , and denote lookup by $\Gamma(x)$. Then $\Gamma[N \mapsto V]$ denotes the mapping such that $\Gamma[N \mapsto V](N) = V$ and $\Gamma[N \mapsto V](x) = \Gamma(x)$ if $x \neq N$. Denote the empty mapping by ε , lists of bindings by $[\overline{N} \mapsto \overline{V}]$ and $[\overline{N} \mapsto \overline{V}, \overline{N}' \mapsto \overline{V}']$, and mapping composition by $\Gamma \circ \Gamma'$ (where $\Gamma \circ \Gamma'(x) = \Gamma'(x)$ if $x \in \text{dom}(\Gamma')$ and $\Gamma \circ \Gamma'(x) = \Gamma(x)$ otherwise). We say that Γ' *extends* Γ , denoted $\Gamma \subseteq \Gamma'$, if $\text{dom}(\Gamma) \subseteq \text{dom}(\Gamma')$ and $\Gamma(x) = \Gamma'(x)$ for $x \in \text{dom}(\Gamma)$.

$$\begin{array}{c}
\text{(T-CONSDECL)} \\
\frac{\Gamma(\text{Co}) = \bar{A} \rightarrow D[\langle \bar{B} \rangle]}{\Gamma \vdash \text{Co}(\bar{A}) : D[\langle \bar{B} \rangle]}
\end{array}
\quad
\begin{array}{c}
\text{(T-DATADECL)} \\
\frac{\Gamma \vdash \overline{\text{Cons}} : D[\langle \bar{A} \rangle]}{\Gamma \vdash \mathbf{data} D[\langle \bar{A} \rangle] = \overline{\text{Cons}};}
\end{array}
\quad
\begin{array}{c}
\text{(T-SUB)} \\
\frac{\Gamma \vdash e : T \quad T \preceq T'}{\Gamma \vdash e : T'}
\end{array}$$

$$\begin{array}{c}
\text{(T-BOOL)} \\
\Gamma \vdash b : \mathbf{Bool}
\end{array}
\quad
\begin{array}{c}
\text{(T-NULL)} \\
\Gamma \vdash \mathbf{null} : A
\end{array}
\quad
\begin{array}{c}
\text{(T-FUNCEXPR)} \\
\frac{\text{tmatch}(\bar{A}, \bar{C}) = \sigma \quad \sigma \neq \perp}{\Gamma \vdash \bar{e} : \bar{C} \quad \Gamma(\text{fn}) = \bar{A} \rightarrow B}$$

$$\begin{array}{c}
\text{(T-CONSEXPR)} \\
\frac{\Gamma \vdash \bar{e} : \bar{C} \quad \sigma \neq \perp \quad \text{tmatch}(\bar{A}, \bar{C}) = \sigma}{\Gamma(\text{Co}) = \bar{A} \rightarrow D[\langle \bar{B} \rangle]}$$

$$\begin{array}{c}
\text{(T-WILDCARD)} \\
\Gamma \vdash _ : A
\end{array}
\quad
\begin{array}{c}
\text{(T-VAR)} \\
\frac{\Gamma(x) = A}{\Gamma \vdash x : A}
\end{array}$$

$$\begin{array}{c}
\text{(T-FUNCDECL)} \\
\frac{\Gamma(\text{fn}) = \bar{B} \rightarrow C \quad \Gamma[\bar{x} \mapsto \bar{B}] \vdash e : C}{\Gamma \vdash \mathbf{def} C \text{ fn}[\langle \bar{A} \rangle](\bar{B} \ x) = e;}
\end{array}
\quad
\begin{array}{c}
\text{(T-BRANCH)} \\
\frac{\Gamma' \vdash p : A \quad \Gamma' \vdash e : B \quad \Gamma' = \Gamma \circ \text{psubst}(p, A, \Gamma)}{\Gamma \vdash p \Rightarrow e : A \rightarrow B}
\end{array}
\quad
\begin{array}{c}
\text{(T-CASE)} \\
\frac{\Gamma \vdash e : A \quad \Gamma \vdash \overline{\text{br}} : A \rightarrow B}{\Gamma \vdash \mathbf{case} e \{ \overline{\text{br}} \} : B}
\end{array}$$

Fig. 3. The type system for the functional level of ABS

A *typing context* Γ is a mapping from names to typings which assigns types A to variables, type constants T to constants, and type signatures $\bar{A} \rightarrow B$ to function symbols. For simplicity, overloading is not considered. A name can only have one typing, and interface and class names are assumed to be distinct. We omit the typing of basic types such as \mathbf{Bool} and \mathbf{Int} , and assume that expressions of the basic types are type checked directly as in the rule T- \mathbf{Bool} in Fig. 3.

The *Functional Level of the ABS Type System* is shown in Fig. 3. We assume a typing context Γ which maps names to their declared types; i.e., the initial typing context gives types to variables and to (user-defined) constructors and functions. The expression \mathbf{null} can have any type by T-NULL. A variable is well-typed if declared in Γ by T-VAR. In T-CONSDECL, constructor declarations are treated like variables. (Note that the constructor may be parametric; e.g., for $\mathbf{List}(A)$, the list constructor Cons should have the type $A, \mathbf{List}(A) \rightarrow \mathbf{List}(A)$.) In T-CONSEXPR, a constructor expression is well-typed if its actual and formal parameter types are the same when matching the type variables of the formal parameter type to the actual parameter types by the auxiliary function tmatch . If there is no match, $\text{tmatch}(\bar{A}, \bar{C})$ returns \perp . (For example, if x is an \mathbf{Int} and y is a $\mathbf{List}(\mathbf{Int})$, then $\text{Cons}(x, y)$ should get type $\mathbf{List}(\mathbf{Int})$, which happens since tmatch binds A to \mathbf{Int} .) Function definition and application are handled in the same way by T-FUNCDECL and T-FUNCEXPR. Additionally the function body is type-checked in Γ extended with the typing of formal parameters in T-FUNCDECL, which may again be type variables.

The declaration of a data type is well-typed if its constructors are well-typed, by T-DATADECL. Case expressions are well-typed by T-CASE if all branches type check to the same type. The pattern must have the same type A as the case expression. A branch $p \Rightarrow e$ is well-typed by T-BRANCH if there is an extension of Γ which adds types for the new variables in the pattern p and which allows the expression e to be type-checked. The desired mapping can be constructed from A and p by induction over the structure of p as follows: If A is a type variable, then p is a variable and $\text{psubst}(p, A, \Gamma) = [p \mapsto A]$. Otherwise, we proceed by induction

$$\begin{array}{c}
\text{(T-POLL)} \\
\frac{\Gamma \vdash e : \mathbf{fut}\langle T \rangle}{\Gamma \vdash e? : \mathbf{Bool}}
\end{array}
\quad
\begin{array}{c}
\text{(T-GET)} \\
\frac{\Gamma \vdash x : \mathbf{fut}\langle T \rangle}{\Gamma \vdash x.\mathbf{get} : T}
\end{array}
\quad
\begin{array}{c}
\text{(T-SKIP)} \\
\frac{}{\Gamma \vdash \mathbf{skip}}
\end{array}
\quad
\begin{array}{c}
\text{(T-AWAIT)} \\
\frac{\Gamma \vdash g : \mathbf{Bool}}{\Gamma \vdash \mathbf{await } g}
\end{array}
\quad
\begin{array}{c}
\text{(T-SUSPEND)} \\
\frac{}{\Gamma \vdash \mathbf{suspend}}
\end{array}$$

$$\begin{array}{c}
\text{(T-COMPOSITION)} \\
\frac{\Gamma \vdash s \quad \Gamma \vdash s'}{\Gamma \vdash s; s'}
\end{array}
\quad
\begin{array}{c}
\text{(T-ASSIGN)} \\
\frac{\Gamma \vdash rhs : \Gamma(v)}{\Gamma \vdash v = rhs}
\end{array}
\quad
\begin{array}{c}
\text{(T-AND)} \\
\frac{\Gamma \vdash g_1 : \mathbf{Bool} \quad \Gamma \vdash g_2 : \mathbf{Bool}}{\Gamma \vdash g_1 \wedge g_2 : \mathbf{Bool}}
\end{array}
\quad
\begin{array}{c}
\text{(T-NEW)} \\
\frac{\Gamma \vdash \bar{e} : \text{ptypes}(C) \quad T \in \text{interfaces}(C)}{\Gamma \vdash \mathbf{new} [\mathbf{cog}] C(\bar{e}) : T}
\end{array}$$

$$\begin{array}{c}
\text{(T-ASYNC CALL)} \\
\frac{\Gamma \vdash e.m(\bar{e}) : T}{\Gamma \vdash e!m(\bar{e}) : \mathbf{fut}\langle T \rangle}
\end{array}
\quad
\begin{array}{c}
\text{(T-CONDITIONAL)} \\
\frac{\Gamma \vdash b : \mathbf{Bool} \quad \Gamma \vdash s_1 \quad \Gamma \vdash s_2}{\Gamma \vdash \mathbf{if } b \{s_1\} \mathbf{else} \{s_2\}}
\end{array}
\quad
\begin{array}{c}
\text{(T-WHILE)} \\
\frac{\Gamma \vdash b : \mathbf{Bool} \quad \Gamma \vdash s}{\Gamma \vdash \mathbf{while } b \{s\}}
\end{array}$$

$$\begin{array}{c}
\text{(T-RETURN)} \\
\frac{\Gamma \vdash e : T \quad \Gamma(\text{destiny}) = \mathbf{fut}\langle T \rangle}{\Gamma \vdash \mathbf{return } e}
\end{array}
\quad
\begin{array}{c}
\text{(T-SYNC CALL)} \\
\frac{\Gamma \vdash e : N \quad \Gamma \vdash \bar{e} : \bar{T} \quad \text{match}(m, \bar{T} \rightarrow T, N)}{\Gamma \vdash e.m(\bar{e}) : T}
\end{array}
\quad
\begin{array}{c}
\text{(T-METHOD)} \\
\frac{\Gamma' = \Gamma[\bar{x} \mapsto \bar{T}, x' \mapsto \bar{T}'] \quad \Gamma'[\text{destiny} \mapsto \mathbf{fut}\langle T'' \rangle] \vdash s}{\Gamma \vdash T'' m(\bar{T} \bar{x})\{T' x'; s\}}
\end{array}$$

$$\begin{array}{c}
\text{(T-CLASS)} \\
\frac{\forall I \in \bar{I} \cdot \text{implements}(C, I) \quad \Gamma[\text{this} \mapsto C, \text{fields}(C)] \vdash \bar{M}}{\Gamma \vdash \mathbf{class } C \mathbf{implements } \bar{I} \{ \bar{T} \bar{f}; \bar{M} \}}
\end{array}
\quad
\begin{array}{c}
\text{(T-PROGRAM)} \\
\frac{\Gamma[\bar{x} \mapsto \bar{T}] \vdash s \quad \forall Dd \in \bar{Dd} \cdot \Gamma \vdash Dd \quad \forall CL \in \bar{CL} \cdot \Gamma \vdash CL \quad \forall F \in \bar{F} \cdot \Gamma \vdash F}{\Gamma \vdash \bar{Dd} \bar{F} \bar{IF} \bar{CL} \{ \bar{T} \bar{x}; s \}}
\end{array}$$

Fig. 4. The type system for the concurrent object level of ABS

over p . If $p = x$, $psubst(p, A, \Gamma) = \text{if } \Gamma(x) = T \text{ then } \varepsilon \text{ else } [p \mapsto A] \text{ fi}$. If $p = t$ or $p = _$, $psubst(p, A, \Gamma) = \varepsilon$. Otherwise $p = Co(p_1, \dots, p_n)$ such that $\Gamma(Co) = A_1, \dots, A_n \rightarrow A$, and $psubst(p, A, \Gamma) = psubst(p_1, A_1) \circ \dots \circ psubst(p_n, A_n, \Gamma)$. The type of a variable x in p must be the same as in $\Gamma(x)$, unless it is new.

Subtyping. $T \preceq T'$ is nominal and reflects the extension relation on interfaces. For simplicity we extend the subtype relation such that $C \preceq I$ if class C implements interface I ; object identifiers are typed by their class and object references by their interface. We don't consider subtyping for data types or type variables.

The Concurrent Object Level of the ABS Type System is given in Fig. 4. By T-PROGRAM, a program is well-typed if its data types, functions, interfaces, classes, and its main block are well-typed (we ignore the straightforward type checking of interface declarations). By T-CLASS, a class is well-typed if its methods are well-typed in the typing context extended by the typing of its fields. We add a fresh name *this* to the typing context, which is typed by C , allowing internal methods to be invoked. We omit the definitions of the auxiliary functions of the type system, which are straightforward; e.g., $fields(C)$ returns the typing context given by the attributes of C . By T-METHOD, a method declaration is well-typed if its body is well-typed in the typing context extended by the typing of formal parameters and local variables. We add a fresh name *destiny* to the typing context, which binds to the type of the method's future. The rules for **skip**, **suspend**, assignment, composition, conditional, and **while** are standard.

By T-RETURN, a **return** statement from an asynchronous method call is well-typed if its expression types to the type of the method's future. By T-AWAIT,

$$\begin{array}{ll}
cn ::= \epsilon \mid fut \mid object \mid invoc \mid cog \mid cn \ cn & cog ::= cog(c, act) \\
fut ::= fut(f, val) & val ::= v \mid \perp \\
object ::= ob(o, a, p, q) & a ::= T \ x \ v \mid a, a \\
process ::= \{a \mid s\} \mid \mathbf{error} & p ::= process \mid \mathbf{idle} \\
q ::= \epsilon \mid process \mid q \ q & v ::= o \mid f \mid b \mid t \\
invoc ::= invoc(o, f, m, \bar{v}) & act ::= o \mid \epsilon \\
s ::= \mathbf{cont}(f) \mid \dots
\end{array}$$

Fig. 5. Runtime syntax; here, o , f , and c are object, future, and cog identifiers

await g is well-typed if g is of type **Bool**, rule T-AND decomposes guards, and by T-POLL a reply-guard $e?$ is a **Bool** if e is a future reference. Similarly, by T-GET, the **get** operation unwraps the type of a future. By T-NEW, object creation has a type T if the actual parameters can be typed to the types of the formal parameters (given by a function $ptypes$) and T is among the declared interfaces of the class. By T-ASYNC CALL, an asynchronous method call has type **fut** $\langle T \rangle$ if the corresponding synchronous call has type T . By T-SYNC CALL, a call to a method m has type T if its actual parameters have types \bar{T} and the signature $\bar{T} \rightarrow T$ matches a signature for m in the known interface of the callee (given by an auxiliary function $match$).

5 An Operational Semantics for Core ABS

The operational semantics of ABS is presented as a transition system in an SOS style [33]. Rules apply to subsets of configurations (the standard context rules are not listed). For simplicity we assume that configurations can be reordered to match the left-hand side of the rules (i.e., matching is modulo associativity and commutativity as in rewriting logic [30]). A run is a possibly nonterminating sequence of rule applications. When auxiliary functions are used in the semantics, these are evaluated in between the applications of transition rules in a run.

5.1 Runtime Configurations

The runtime syntax is given in Fig. 5. *Configurations* cn are sets of objects, invocation messages, concurrent object groups (cogs), and futures. The associative and commutative union operator on configurations is denoted by whitespace and the empty configuration by ϵ . Configurations are written inside curly brackets; in the term $\{cn\}$, cn captures the *entire* configuration. A *substitution* is a mapping from variable names to values (for convenience, we associate the declared type of the variable with the binding). An *object* is a term $ob(o, a, p, q)$ where o is the object's identifier, a a substitution representing the object's fields, p an *active process*, and q a *pool of suspended processes*. A process p consists of a substitution l of local variable bindings and a list s of statements, denoted by $\{l \mid s\}$ when convenient or it is *idle*. The statement $\mathbf{cont}(f)$ is used to control scheduling when local synchronous calls complete their execution, returning control to the caller. In an *invocation message* $invoc(o, f, m, \bar{v})$, o is the callee, f the future to

$$\begin{array}{c}
\text{(REDCONS)} \\
\frac{\sigma \vdash e_i \rightsquigarrow \sigma' \vdash e'_i \quad 1 \leq i \leq n}{\sigma \vdash Co(e_1, \dots, e_i, \dots, e_n)} \\
\rightsquigarrow \sigma' \vdash Co(e_1, \dots, e'_i, \dots, e_n)
\end{array}
\qquad
\begin{array}{c}
\text{(REDFUNEXP)} \\
\frac{\sigma \vdash e_i \rightsquigarrow \sigma' \vdash e'_i \quad 1 \leq i \leq n}{\sigma \vdash fn(e_1, \dots, e_i, \dots, e_n)} \\
\rightsquigarrow \sigma' \vdash fn(e_1, \dots, e'_i, \dots, e_n)
\end{array}$$

$$\begin{array}{c}
\text{(REDVAR)} \\
\sigma \vdash x \rightsquigarrow \sigma \vdash \sigma(x)
\end{array}
\qquad
\begin{array}{c}
\text{(REDCASE1)} \\
\frac{\sigma \vdash e \rightsquigarrow \sigma' \vdash e'}{\sigma \vdash \mathbf{case} \ e \ \{br\}} \\
\rightsquigarrow \sigma' \vdash \mathbf{case} \ e' \ \{br\}
\end{array}
\qquad
\begin{array}{c}
\text{(REDCASE3)} \\
\frac{\text{match}(\sigma(p), t) = \perp}{\sigma \vdash \mathbf{case} \ t \ \{p \Rightarrow e; br\}} \\
\rightsquigarrow \sigma \vdash \mathbf{case} \ t \ \{br\}
\end{array}$$

$$\begin{array}{c}
\text{(REDFUNGROUND)} \\
\frac{\text{fresh}(\{y_1, \dots, y_n\}) \\
\bar{y} = y_1, \dots, y_n \\
\text{length}(\bar{x}_{fn}) = n}{\sigma \vdash fn(\bar{t})} \\
\rightsquigarrow \sigma[\bar{y} \mapsto \bar{t}] \vdash e_{fn}[\bar{x}_{fn} \mapsto \bar{y}]
\end{array}
\qquad
\begin{array}{c}
\text{(REDCASE2)} \\
\frac{\text{match}(\sigma(p), t) \neq \perp \\
\sigma' = \sigma[y_i \mapsto \text{match}(\sigma(p), t)(x_i)] \ \text{for } 1 \leq i \leq n \\
\bar{y} = y_1, \dots, y_n \quad \text{fresh}(\{y_1, \dots, y_n\}) \\
\bar{x} = x_1, \dots, x_n \quad \{x_1, \dots, x_n\} = \text{vars}(\sigma(p))}{\sigma \vdash \mathbf{case} \ t \ \{p \Rightarrow e; br\} \rightsquigarrow \sigma' \vdash e[\bar{x} \mapsto \bar{y}]}
\end{array}$$

Fig. 6. The evaluation of functional expressions

which the call's result is returned, m the method name, and \bar{v} the call's actual parameter values. A *cog* only contains an identifier c and the currently active object o , or ϵ if no object of the cog is currently active (i.e., all objects have the idle process as active process). A *future* $fut(f, v)$ has an identifier f and a reply value v (which is \perp when the future's reply value has not been received). Values are object and future identifiers, Boolean values, and ground terms from the functional language. For simplicity, classes are not represented explicitly in the semantics, as they may be seen as static tables.

5.2 A Reduction System for ABS Functional Expressions

The evaluation of functional expressions is defined by the small-step reduction relation $\sigma \vdash e \rightsquigarrow \sigma' \vdash e'$, given in Fig. 6, which reduces an expression e in the context of a *substitution* σ to e' in the context of σ' . A substitution σ is *well-typed* in a typing context Γ , denoted $\Gamma \vdash \sigma$, if $\Gamma \vdash \sigma(x) : \Gamma(x)$ for all $x \in \text{dom}(\sigma)$.

Let $e[\bar{x} \mapsto \bar{y}]$ denote the expression e in which every occurrence of x_i has been replaced by the corresponding y_i . The predicate $\text{fresh}(\{x_1, \dots, x_j\})$ asserts that any variable name x_i (for $1 \leq i \leq j$) is globally unique. Let the syntactic category t consist of ground terms, i.e., constructor terms with only ground terms in argument positions, built-in constants such as **null**, and object names.

Function evaluation is strict. For a function fn defined by **def** $T \ fn(\bar{T} \ \bar{x}) = e$, denote by \bar{x}_{fn} the formal parameter list \bar{x} and by e_{fn} the body e . The evaluation of a function call $fn(\bar{e})$ in a context σ reduces to the evaluation of $e_{fn}[\bar{x}_{fn} \mapsto \bar{y}]$ in the context $\sigma[\bar{y} \mapsto \bar{t}]$ after the arguments \bar{e} have successfully been reduced to ground terms \bar{t} . The change in scope for evaluating a function body is obtained by replacing the formal parameters \bar{x}_{fn} with *fresh variables* \bar{y} in the function body, thus avoiding name capture while keeping the full context σ .

Case expressions will only be reduced if the pattern in one of the branches matches. The function $\text{match}(p, t)$ returns the unique substitution σ such that

$\sigma(p) = t$ and $\text{dom}(\sigma) = \text{vars}(p)$ (otherwise, $\text{match}(p, t) = \perp$). Note how in REDCASE2, the current substitution σ is applied to the pattern before matching, which allows the pattern to first match with the current state. For pattern matching, variables in the pattern p are bound to ground terms in the term t , applying a similar variable renaming as for function evaluation. Thus the context for evaluating the right-hand side e of the branch $p \Rightarrow e$ extends the current substitution σ with the bindings that occurred during the pattern matching.

The variable renaming in the rules which change the scope of variables, i.e., REDFUNGROUND and REDCASE2, allows small-step reductions in the arguments to constructors and function application in REDCONS and REDFUNEXP, since the additional variables will not introduce name conflicts.

Lemma 1 (Type preservation). *Let Γ be a typing context and σ a substitution such that $\Gamma \vdash \sigma$. If $\Gamma \vdash e : A$ and $\sigma \vdash e \rightsquigarrow \sigma' \vdash e'$, then there is a typing context Γ' such that $\Gamma \subseteq \Gamma'$, $\Gamma' \vdash \sigma'$, and $\Gamma' \vdash e' : A$.*

Proof. The proof is by induction over the application of reduction rules.

- REDVAR. By assumption, $\Gamma \vdash \sigma$ and $\Gamma \vdash x : A$. Since σ is well-typed, $\Gamma \vdash \sigma(x) : \Gamma(x)$, so $\Gamma \vdash \sigma(x) : A$.
- REDCONS. By the induction hypothesis (IH), $\Gamma \vdash e_i : A_i$, $\Gamma' \vdash e'_i : A_i$, $\Gamma \subseteq \Gamma'$, and $\Gamma' \vdash \sigma'$. By assumption, $\Gamma \vdash \text{Co}(e_1, \dots, e_i, \dots, e_n) : A$. Since $\Gamma \subseteq \Gamma'$, $\Gamma' \vdash \text{Co}(e_1, \dots, e'_i, \dots, e_n) : A$.
- REDFUNEXP. Similar to the case for REDCONS.
- REDFUNGROUND. By assumption, $\Gamma \vdash \sigma$, $\Gamma \vdash \text{fn}(\bar{t}) : A$, and $\Gamma \vdash t_i : A_i$ for all t_i in \bar{t} . Thus, we may assume a function declaration for fn such that $\Gamma(\text{fn}) = \bar{T} \rightarrow T'$ and a type substitution ρ such that $T'\rho = A$ and $T_i\rho = A_i$ for all T_i in \bar{T} . Obviously, $\Gamma[\bar{x}_{\text{fn}} \mapsto \bar{T}\rho] \vdash [\bar{x}_{\text{fn}} \mapsto \bar{T}]$. By T-FUNCDECL, $\Gamma[\bar{x}_{\text{fn}} \mapsto \bar{T}] \vdash e_{\text{fn}} : T'$. Typing is preserved under type substitutions [32], so $\Gamma[\bar{x}_{\text{fn}} \mapsto \bar{T}\rho] \vdash e_{\text{fn}} : T'\rho$, which is the same as $\Gamma[\bar{x}_{\text{fn}} \mapsto \bar{A}_i] \vdash e_{\text{fn}} : A$. After variable renaming, we let $\Gamma' = \Gamma[\bar{y} \mapsto \bar{A}_i]$ and get $\Gamma' \vdash e_{\text{fn}}[\bar{x}_{\text{fn}} \mapsto \bar{y}] : A$. Since $\{y_1, \dots, y_n\}$ are fresh, we have $\Gamma \subseteq \Gamma'$ and $\Gamma' \vdash \sigma$, so $\Gamma' \vdash \sigma'$.
- REDCASE1. By assumption $\Gamma \vdash e : T$, and by the IH $\Gamma \subseteq \Gamma'$, $\Gamma' \vdash \sigma'$, and $\Gamma' \vdash e' : T$. Since $\Gamma \subseteq \Gamma'$, $\Gamma' \vdash \mathbf{case} e' \{\bar{br}\}$.
- REDCASE2. By assumption, $\Gamma \vdash \sigma$, $\Gamma \vdash \mathbf{case} t \{p \Rightarrow e; \bar{br}\} : A$, and $\text{match}(\sigma(p), t) \neq \perp$. Since we match with $\sigma(p)$, $\text{vars}(\sigma(p)) \cap \text{dom}(\sigma) = \emptyset$. By T-CASE, there is some type T such that $\Gamma \vdash t : T$ and $\Gamma \vdash p \Rightarrow e : T \rightarrow A$. By T-BRANCH, there is a type substitution $\rho = \text{psubst}(\sigma(p), T) \neq \perp$ such that for $\Gamma'' = \Gamma \circ \rho$, $\Gamma'' \vdash \sigma(p) : T$ and $\Gamma'' \vdash e : A$. Since $\text{dom}(\rho) \cap \text{dom}(\sigma) = \emptyset$, $\Gamma'' \vdash \sigma \circ \text{match}(\sigma(p), t)$. Renaming the variables in $\sigma(p)$, we define $\Gamma' = \Gamma[y_i \mapsto \Gamma''(x_i)]$ for $1 \leq i \leq n$. Obviously, $\Gamma \subseteq \Gamma'$. Since y_1, \dots, y_n are fresh, renaming variables uniformly does not change the derivations, so we get $\Gamma' \vdash \sigma'$ and $\Gamma' \vdash e[\bar{x} \mapsto \bar{y}] : A$.
- REDCASE3. Since $\Gamma \vdash \mathbf{case} t \{p \Rightarrow e; \bar{br}\} : A$, we have $\Gamma \vdash \mathbf{case} t \{\bar{br}\} : A$. □

It follows from Lemma 1 that given a well-typed expression e and a well-typed context σ , then all states in the reduction sequence from $\sigma \vdash e$ will be well-typed,

(REDBOOLGUARD)	(REDREPLYGUARD1)	(REDREPLYGUARD2)	(REDGUARDS)
$\sigma \vdash b$	$\sigma \vdash e \rightsquigarrow \sigma \vdash f$	$\sigma \vdash e \rightsquigarrow \sigma \vdash f$	$\sigma, cn \vdash g_1 \rightsquigarrow \sigma, cn \vdash g'_1$
$\rightsquigarrow \sigma \vdash b'$	$\text{fut}(f, v) \in cn \quad v \neq \perp$	$\text{fut}(f, \perp) \in cn$	$\sigma, cn \vdash g_2 \rightsquigarrow \sigma, cn \vdash g'_2$
$\sigma, cn \vdash b$	$\sigma, cn \vdash e?$	$\sigma, cn \vdash e?$	$\sigma, cn \vdash g_1 \wedge g_2$
$\rightsquigarrow \sigma, cn \vdash b'$	$\rightsquigarrow \sigma, cn \vdash \text{true}$	$\rightsquigarrow \sigma, cn \vdash \text{false}$	$\rightsquigarrow \sigma, cn \vdash g'_1 \wedge g'_2$

Fig. 7. The evaluation of guard expressions

independent of the order of reductions. If an expression e in a context σ reduces to a ground term t , we denote the resulting value by $\llbracket e \rrbracket_\sigma$. This value, however, is not guaranteed to exist, for two reasons: first, the reduction sequence might not terminate; second, the normal form may not be a ground term, because branches in case expressions need not have full coverage. (We assume here that normal forms are unique, although we do not prove this.) In either case, we know by Lemma [11](#) that all states in such a reduction sequence are well-typed.

5.3 The Operational Semantics for Concurrent Objects in ABS

Evaluating Guards. Given a substitution σ and a configuration cn , we lift the reduction relation for functional expressions to guards by the rules of Fig. [7](#). It follows from Lemma [11](#) that well-typedness is preserved by guard reduction in the context of well-typed substitutions and configurations. If a guard g in a context σ, cn reduces to a ground term b , we denote the resulting value by $\llbracket g \rrbracket_\sigma^{cn}$.

Auxiliary functions. If T is the return type of a method m in a class C , we let $\text{bind}(o, f, m, \bar{v}, C)$ return a process resulting from the activation of m in C with actual parameters \bar{v} , callee o and associated future f . If binding succeeds, this process has a local variable *destiny* of type $\text{fut}\langle T \rangle$ bound to f , and the formal parameters are bound to \bar{v} . If binding does *not* succeed, we get the **error** process. The function $\text{atts}(C, \bar{v}, o, c)$ returns the initial state of an instance of class C , in which the formal parameters are bound to \bar{v} and the reserved variables *this* and *cog* are bound to the object and cog identities o and c , respectively. The function $\text{init}(C)$ returns an activation of the *init* method of C , if defined, and otherwise the *idle* process. The predicate $\text{fresh}(n)$ asserts that a name n is globally unique (where n may be an identifier for an object, a future, or a cog).

Transition rules transform state configurations into new configurations, and are given in Figs. [8](#) and [9](#). There are different assignment rules for functional expressions (ASSIGN-LOCAL and ASSIGN-FIELD), object creation (NEW-OBJECT and NEW-COG-OBJECT), method calls (ASYNC-CALL, COG-SYNC-CALL and SELF-SYNC-CALL), and future dereferencing (READ-FUT). Rule SKIP consumes a **skip** in the active process. Here and in the sequel, the variable s will match any (possibly empty) statement list. Rules ASSIGN-LOCAL and ASSIGN-FIELD assign the value of expression e to a variable x in the local variables l or in the fields a , respectively. Rules COND-TRUE and COND-FALSE branch the execution depending on the value obtained by evaluating the expression e . (We omit the standard rule for **while**.)

Process Suspension and Activation. Three operations manipulate a process pool q : $q \cup p$ adds a process p to q , $q \setminus p$ removes p from q , and $\text{select}(q, a, cn)$ selects

$$\begin{array}{c}
\text{(SKIP)} \\
\frac{}{ob(o, a, \{l|\mathbf{skip}; s\}, q) \rightarrow ob(o, a, \{l|s\}, q)} \\
\\
\text{(COND-TRUE)} \\
\frac{\llbracket e \rrbracket_{(aol)}}{ob(o, a, \{l|\mathbf{if } e \mathbf{ then } \{s_1\} \mathbf{ else } \{s_2\}; s\}, q) \rightarrow ob(o, a, \{l|s_1; s\}, q)} \\
\\
\text{(SUSPEND)} \\
\frac{}{ob(o, a, \{l|\mathbf{suspend}; s\}, q) \rightarrow ob(o, a, \text{idle}, q \cup \{l|s\})} \\
\\
\text{(ACTIVATE)} \\
\frac{p = \text{select}(q, a, cn) \quad c = a(\text{cog})}{\{ob(o, a, \text{idle}, q) \text{ cog}(c, \epsilon) \text{ cn}\} \rightarrow \{ob(o, a, p, q \setminus p) \text{ cog}(c, o) \text{ cn}\}} \\
\\
\text{(ASYN-CALL)} \\
\frac{o' = \llbracket e \rrbracket_{(aol)} \quad \bar{v} = \llbracket \bar{e} \rrbracket_{(aol)} \quad \text{fresh}(f)}{ob(o, a, \{l|x = e!m(\bar{e}); s\}, q) \rightarrow ob(o, a, \{l|x = f; s\}, q) \text{ invoc}(o', f, m, \bar{v}) \text{ fut}(f, \perp)} \\
\\
\text{(ASSIGN-LOCAL)} \\
\frac{x \in \text{dom}(l) \quad v = \llbracket e \rrbracket_{(aol)}}{ob(o, a, \{l|x = e; s\}, q) \rightarrow ob(o, a, \{l|x \mapsto v\}|s\}, q)} \\
\\
\text{(ASSIGN-FIELD)} \\
\frac{x \in \text{dom}(a) \quad v = \llbracket e \rrbracket_{(aol)}}{ob(o, a, \{l|x = e; s\}, q) \rightarrow ob(o, a[x \mapsto v], \{l|s\}, q)} \\
\\
\text{(COND-FALSE)} \\
\frac{\neg \llbracket e \rrbracket_{(aol)}}{ob(o, a, \{l|\mathbf{if } e \mathbf{ then } \{s_1\} \mathbf{ else } \{s_2\}; s\}, q) \rightarrow ob(o, a, \{l|s_2; s\}, q)} \\
\\
\text{(RELEASE-COG)} \\
\frac{c = a(\text{cog})}{ob(o, a, \text{idle}, q) \text{ cog}(c, o) \rightarrow ob(o, a, \text{idle}, q) \text{ cog}(c, \epsilon)} \\
\\
\text{(AWAIT-TRUE)} \\
\frac{\llbracket g \rrbracket_{(aol)}^{cn}}{\{ob(o, a, \{l|\mathbf{await } g; s\}, q) \text{ cn}\} \rightarrow \{ob(o, a, \{l|s\}, q) \text{ cn}\}} \\
\\
\text{(AWAIT-FALSE)} \\
\frac{\neg \llbracket g \rrbracket_{(aol)}^{cn}}{\{ob(o, a, \{l|\mathbf{await } g; s\}, q) \text{ cn}\} \rightarrow \{ob(o, a, \{l|\mathbf{suspend}; \mathbf{await } g; s\}, q) \text{ cn}\}} \\
\\
\text{(BIND-MTD)} \\
\frac{p' = \text{bind}(o, f, m, \bar{v}, \text{class}(o))}{ob(o, a, p, q) \text{ invoc}(o, f, m, \bar{v}) \rightarrow ob(o, a, p, q \cup p')}
\end{array}$$

Fig. 8. Semantics of the concurrent object level of Core ABS (1)

a process from q (if q is empty or no process is *ready*, the result is the idle process [24]). The actual definitions of these operations are left unspecified; different definitions correspond to different scheduling policies for processes, although care must be taken that `select` always gives the initial process of an object the highest priority (otherwise another process might see uninitialized object states).

Let \emptyset denote the empty pool. Rule `SUSPEND` suspends the active process to the process pool, leaving the processor idle, and `RELEASE-COG` makes the cog idle if its active object is idle. Rule `AWAIT-TRUE` consumes `await g` if g evaluates to true in the object's current state, `AWAIT-FALSE` adds a `suspend` statement to the process if the guard evaluates to false. Rule `ACTIVATE` selects a process p from the process pool for execution if p is *ready* to execute, i.e., if p would not directly be resuspended or block the processor [24]. These rules ensure that a process can only be scheduled if the cog associated with the object is idle, and that an object always acquires the cog if its process is activated. Synchronous calls and synchronous self-calls, which also influence scheduling, are discussed below.

Communication and Object Creation. Rule `ASYN-CALL` sends an invocation message to o' with a new future f (which is unique since $\text{fresh}(f)$), the method name m , and actual parameters \bar{v} . The return value of f is undefined (i.e., \perp). Rule `BIND-MTD` consumes an invocation message and places the process corresponding to the method activation in the callee's process pool. A reserved local variable *destiny* stores the identity of the future associated with the call.

Rule `RETURN` in Fig. 9 places the return value into the call's associated future. Rule `READ-FUT` dereferences the future f if $v \neq \perp$. If $v = \perp$, the reduction on this object is *blocked*. Rules `COG-SYN-CALL` and `COG-SYN-RETURN-SCHED`

$$\begin{array}{c}
\text{(RETURN)} \\
\frac{v = \llbracket e \rrbracket_{(aol)} \quad l(\text{destiny}) = f}{ob(o, a, \{l \mathbf{return} \ e; s\}, q) \ fut(f, \perp)} \\
\rightarrow ob(o, a, \{l | s\}, q) \ fut(f, v)
\end{array}
\qquad
\begin{array}{c}
\text{(READ-FUT)} \\
\frac{v \neq \perp \quad f = \llbracket e \rrbracket_{(aol)}}{ob(o, a, \{l | x = e.\mathbf{get}; s\}, q) \ fut(f, v)} \\
\rightarrow ob(o, a, \{l | x = v; s\}, q) \ fut(f, v)
\end{array}$$

$$\begin{array}{c}
\text{(COG-SYNC-CALL)} \\
\frac{
\begin{array}{c}
o' = \llbracket e \rrbracket_{(aol)} \quad \bar{v} = \llbracket \bar{e} \rrbracket_{(aol)} \quad \text{fresh}(f) \\
a'(\text{cog}) = c \quad f' = l(\text{destiny}) \\
\{l' | s'\} = \text{bind}(o', f, m, \bar{v}, \text{class}(o'))
\end{array}
}{
\begin{array}{c}
ob(o, a, \{l | x = e.m(\bar{e}); s\}, q) \\
ob(o', a', \text{idle}, q') \ cog(c, o)
\end{array}
} \\
\rightarrow ob(o, a, \text{idle}, q \cup \{l | x = f.\mathbf{get}; s\}) \ fut(f, \perp) \\
ob(o', a', \{l' | s'; \mathbf{cont}(f')\}, q') \ cog(c, o')
\end{array}
\qquad
\begin{array}{c}
\text{(COG-SYNC-RETURN-SCHED)} \\
\frac{
\begin{array}{c}
a'(\text{cog}) = c \quad l'(\text{destiny}) = f \\
ob(o, a, \{l | \mathbf{cont}(f)\}, q) \ cog(c, o) \\
ob(o', a', \text{idle}, q' \cup \{l' | s'\}) \\
\rightarrow ob(o, a, \text{idle}, q) \ cog(c, o') \\
ob(o', a', \{l' | s'\}, q')
\end{array}
}{
}
\end{array}$$

$$\begin{array}{c}
\text{(SELF-SYNC-CALL)} \\
\frac{
\begin{array}{c}
f' = l(\text{destiny}) \quad o = \llbracket e \rrbracket_{(aol)} \quad \bar{v} = \llbracket \bar{e} \rrbracket_{(aol)} \\
\text{fresh}(f) \quad \{l' | s'\} = \text{bind}(o, f, m, \bar{v}, \text{class}(o))
\end{array}
}{
\begin{array}{c}
ob(o, a, \{l | x = e.m(\bar{e}); s\}, q) \\
\rightarrow ob(o, a, \{l' | s'; \mathbf{cont}(f')\}, q \cup \{l | x = f.\mathbf{get}; s\}) \ fut(f, \perp)
\end{array}
}
\end{array}$$

$$\begin{array}{c}
\text{(SELF-SYNC-RETURN-SCHED)} \\
\frac{l'(\text{destiny}) = f}{ob(o, a, \{l | \mathbf{cont}(f)\}, q \cup \{l' | s'\})} \\
\rightarrow ob(o, a, \{l' | s\}, q)
\end{array}
\qquad
\begin{array}{c}
\text{(REM-SYNC-CALL)} \\
\frac{
\begin{array}{c}
o' = \llbracket e \rrbracket_{(aol)} \quad \text{fresh}(f) \quad a(\text{cog}) \neq a'(\text{cog}) \\
ob(o, a, \{l | x = e.m(\bar{e}); s\}, q) \ ob(o', a', p, q') \\
\rightarrow ob(o, a, \{l | f = e!m(\bar{e}); x = f.\mathbf{get}; s\}, q) \\
ob(o', a', p, q')
\end{array}
}{
}
\end{array}$$

$$\begin{array}{c}
\text{(NEW-OBJECT)} \\
\frac{
\begin{array}{c}
\text{fresh}(o') \ p = \text{init}(C) \\
a' = \text{atts}(C, \llbracket \bar{e} \rrbracket_{(aol)}, o', c)
\end{array}
}{
\begin{array}{c}
ob(o, a, \{l | x = \mathbf{new} \ C(\bar{e}); s\}, q) \ cog(c, o) \\
\rightarrow ob(o, a, \{l | x = o'; s\}, q) \ cog(c, o) \\
ob(o', a', \text{idle}, \{p\})
\end{array}
}
\end{array}
\qquad
\begin{array}{c}
\text{(NEW-COG-OBJECT)} \\
\frac{
\begin{array}{c}
\text{fresh}(o') \ \text{fresh}(c') \ p = \text{init}(C) \\
a' = \text{atts}(C, \llbracket \bar{e} \rrbracket_{(aol)}, o', c')
\end{array}
}{
\begin{array}{c}
ob(o, a, \{l | x = \mathbf{new \ cog} \ C(\bar{e}); s\}, q) \\
\rightarrow ob(o, a, \{l | x = o'; s\}, q) \\
ob(o', a', p, \emptyset) \ \cog(c', o')
\end{array}
}
\end{array}$$

Fig. 9. Semantics of the concurrent object level of Core ABS (2)

address synchronous method calls between two objects that are in the same cog. For a synchronous call, possession of the cog directly transfers control from the calling object to the callee and back, bypassing the SUSPEND and ACTIVATE rules. A special **cont** instruction is inserted at the end of the statement list of the new process in COG-SYNC-CALL, which is then used to re-activate the caller process in COG-SYNC-RETURN-SCHED. Synchronous self-calls are implemented similarly by SELF-SYNC-CALL and SELF-SYNC-RETURN-SCHED. The cog invariant (only one object with a non-idle process per cog) is maintained by these rules. A synchronous call to an object of another cog is syntactic sugar for an asynchronous call which is immediately followed by a blocking **get** operation, captured by REM-SYNC-CALL.

Rule NEW-OBJECT creates an object with a unique identifier o' . The object's fields are given default values by $\text{atts}(C, \llbracket \bar{e} \rrbracket_{(aol)}, o', c)$, extended with the actual values \bar{e} for the class parameters (evaluated in the context of the creating process), o' for this and with the creating object's cog c . To instantiate the remaining fields, the process p is queued (we assume that this process reduces to idle if $\text{init}(C)$ is unspecified in the class definition, and that it asynchronously calls **run** if the latter is given). Process p is not directly scheduled in order to uphold the cog invariant (ie., only one object per cog is active), hence any scheduling policy must take care to

$$\begin{array}{c}
\text{(T-STATE1)} \\
\frac{\Delta \vdash_R (v) = T}{\Delta \vdash_R T \ v \ \text{val} \ \mathbf{ok}} \\
\\
\text{(T-STATE2)} \\
\frac{\Delta \vdash_R \text{fds} \ \mathbf{ok} \quad \Delta \vdash_R \text{fds}' \ \mathbf{ok}}{\Delta \vdash_R \text{fds} \ \text{fds}' \ \mathbf{ok}} \\
\\
\text{(T-EMPTY)} \\
\Delta \vdash_R \epsilon \ \mathbf{ok} \\
\\
\text{(T-IDLE)} \\
\Delta \vdash_R \mathbf{idle} \ \mathbf{ok} \\
\\
\text{(T-CONT)} \\
\frac{\Delta(f) = \mathbf{fut}\langle T \rangle}{\Delta \vdash \mathbf{cont}(f)} \\
\\
\text{(T-PROCESS-QUEUE)} \\
\frac{\Delta \vdash_R q \ \mathbf{ok} \quad \Delta \vdash_R q' \ \mathbf{ok}}{\Delta \vdash_R q \ q' \ \mathbf{ok}} \\
\\
\text{(T-OBJECT)} \\
\frac{\text{fields}(\Delta(o)) = [\bar{x} \mapsto \bar{T}] \quad \Delta' \vdash \Delta[\bar{x} \mapsto \bar{T}] \quad \Delta' \vdash_R q \ \mathbf{ok}}{\Delta \vdash_R \text{ob}(o, \bar{T} \ x \ \text{val}, p, q) \ \mathbf{ok}} \\
\\
\text{(T-FUTURE)} \\
\frac{\Delta(f) = \mathbf{fut}\langle T \rangle \quad \text{val} \neq \perp \Rightarrow \Delta(\text{val}) = T}{\Delta \vdash_R \text{fut}(f, \text{val}) \ \mathbf{ok}} \\
\\
\text{(T-PROCESS)} \\
\frac{\Delta' \vdash_R \bar{T} \ x \ \text{val} \ \mathbf{ok} \quad \Delta' \vdash_R s \ \mathbf{ok}}{\Delta \vdash_R (\bar{T} \ x \ \text{val}, s) \ \mathbf{ok}} \\
\\
\text{(T-INVOC)} \\
\frac{\Delta(f) = \mathbf{fut}\langle T \rangle \quad \Delta(\bar{v}) = \bar{T} \quad \text{match}(m, \bar{T} \rightarrow T, \Delta(o))}{\Delta \vdash_R \text{invoc}(o, f, m, \bar{v})} \\
\\
\text{(T-CONFIGURATIONS)} \\
\frac{\Delta \vdash_R \text{cn} \ \mathbf{ok} \quad \Delta \vdash_R \text{cn}' \ \mathbf{ok}}{\Delta \vdash_R \text{cn} \ \text{cn}' \ \mathbf{ok}} \\
\\
\text{(T-COG)} \\
\frac{\Delta(c) = \text{cog}}{\Delta \vdash_R \text{cog}(c, \text{act})}
\end{array}$$

Fig. 10. The typing rules for runtime configurations

always schedule an initial process p with highest priority. Rule **NEW-COG-OBJECT** is like **NEW-OBJECT**, except that a fresh cog is created with o' as its (only) active object, and the initial process p is directly scheduled.

6 Subject Reduction for ABS

The *initial state* of a well-typed program consists of an object $\text{ob}(\text{start}, \epsilon, p, \emptyset)$, where the process p corresponds to the activation of the program's main block. A *run* is a sequence of reductions of an initial state according to the rules of the operational semantics. We now show that a run from a well-typed initial configuration will maintain well-typed configurations; in particular, substitutions remain well-typed and method binding does not result in the **error** process.

Runtime Configurations. Typing rules are given for the runtime syntax shown in Fig. 5. The typing context of the runtime configurations extends the static typing context with types for dynamically created values, i.e., object and future identifiers. Object identifiers are typed by the class of the created object.

Typing Rules for Runtime Configurations. Let $\Delta \vdash_R \text{config} \ \mathbf{ok}$ express that the configuration config is well-typed in the typing context Δ . The typing rules for runtime configurations are given in Fig. 10. In rule **T-OBJECT**, the premise $\text{fields}(\Delta(o)) = [\bar{x} \mapsto \bar{T}]$ asserts that the object's fields have the types declared in its class. If a configuration is well-typed in a typing context Δ , the substitutions a and l (for any object and any process) are well-typed in Δ . Consequently, by Lemma 1, expression and guard evaluation in ABS processes preserves typing.

Well-typedness Assumptions for Auxiliary Functions. Let C be a class with formal parameters \bar{x} of types \bar{T} . We assume that $\text{init}(C)$ returns a well-typed process, and $\text{atts}(C, \bar{v}, o, c)$ a well-typed substitution if \bar{v} have types \bar{T} and o and c are object and cog identifiers, respectively. If C implements a method m with return type T and formal parameters \bar{x}' of types \bar{T}' , let $\text{bind}(o, f, m, \bar{v}', C)$ return a well-typed process if f has type $\mathbf{fut}\langle T \rangle$ and \bar{v}' have the types \bar{T}' .

We prove that the object corresponding to the main block of a well-typed program is well-typed (Lemma 2) and show that the well-typedness of runtime configuration is preserved by reductions (Theorem 1).

Lemma 2. *Let $P \{\overline{T} \overline{x}; s\}$ be an ABS program. If $\Gamma \vdash P \{\overline{T} \overline{x}; s\}$ for some typing context Γ , then $\Gamma \vdash_R \text{ob}(\text{start}, \varepsilon, \{\overline{T} \overline{x} \text{atts}(\overline{T})|s\}, \emptyset) \mathbf{ok}$.*

Proof. Let $\Gamma' = \Gamma[\overline{x} \mapsto \overline{T}]$. It is obvious that $\Gamma' \vdash_R \overline{T} \overline{x} \text{atts}(\overline{T}) \mathbf{ok}$. By assumption, $\Gamma \vdash P \{\overline{T} \overline{x}; s\}$, so $\Gamma' \vdash_R s$. \square

Theorem 1 (Subject Reduction). *If $\Delta \vdash_R cn \mathbf{ok}$ and $cn \rightarrow cn'$, then there is a Δ' such that $\Delta \subseteq \Delta'$ and $\Delta' \vdash_R cn' \mathbf{ok}$.*

Proof. The proof is by induction over the application of transition rules. We assume that class definitions are well-typed (and omit them from the runtime syntax since they do not change). Objects, futures, and messages not affected by a transition remain well-typed, and are ignored below. It follows from Lemma 1 that the reduction of an expression in a well-typed object results in a well-typed object. The transition rules apply when these reductions terminate, reducing an expression e in the state σ to the ground term $\llbracket e \rrbracket_\sigma$. Hence, the reduction of expressions e in states σ occur as $\llbracket e \rrbracket_\sigma$ in the assumptions to the transition rules, and similarly for the evaluation $\llbracket g \rrbracket_\sigma^{cn}$ of guards g in a configuration cn .

- **SKIP.** If $\Delta \vdash_R \text{ob}(o, a, \{l|\mathbf{skip}; s\}, q) \mathbf{ok}$, then $\Delta \vdash_R \text{ob}(o, a, \{l|s\}, q) \mathbf{ok}$.
- **Assignment.** Let $\Delta \vdash_R \text{ob}(o, \overline{T}_1 \overline{x}_1 \overline{v}_1, \{\overline{T}_2 \overline{x}_2 \overline{v}_2|x = e; s\}, q) \mathbf{ok}$. Let $\Delta' = \Delta[\overline{x}_1 \mapsto \overline{T}_1, \overline{x}_2 \mapsto \overline{T}_2]$. Then $\Delta' \vdash x = e; s$, so $\Delta' \vdash e : \Delta'(x)$. Assume that $v = \llbracket e \rrbracket_{aol}$. For ASSIGN-LOCAL, we need to show $\Delta \vdash_R \text{ob}(o, \overline{T}_1 \overline{x}_1 \overline{v}_1, \{\overline{T}_2 \overline{x}_2 \overline{v}_2[x \mapsto v]|x = e; s\}, q) \mathbf{ok}$, which follows from Lemma 1 since $\Delta' \vdash v : \Delta'(x)$. Similarly for ASSIGN-FIELD.
- **Conditionals.** Let $\Delta \vdash_R \text{ob}(o, a, \{l|\mathbf{if} e \{s_1\} \mathbf{else} \{s_s\}; s\}, q) \mathbf{ok}$. By assumption there is a Δ' extending Δ with a and l , such that $\Delta' \vdash e$, $\Delta' \vdash s_1$, $\Delta' \vdash s_2$, and $\Delta' \vdash s$. Consequently, $\Delta' \vdash s_1; s$ and $\Delta' \vdash s_2; s$, and both COND-TRUE and COND-FALSE preserve well-typedness.
- **Process Suspension and Activation.** It is immediate that the rules AWAIT-TRUE, AWAIT-FALSE, ACTIVATE, SUSPEND, RELEASE-COG, SELF-SYNC-RETURN-SCHED, and COG-SYNC-RETURN-SCHED preserve the well-typedness.
- **Object creation.** For NEW-OBJECT, assume $\Delta \vdash_R \text{ob}(o, a, \{l|x = \mathbf{new} C(\overline{e}); s\}, q) \mathbf{ok}$, $\Delta(x) = I$, and *implements*(C, I) (so $C \preceq I$). Since *fresh*(o'), let $\Delta' = \Delta[o' \mapsto C]$. Obviously, $\Delta' \vdash_R \text{ob}(o, a, \{l|x = o'; s\}, q) \mathbf{ok}$. By assumption, a' and p are well-typed in o' , and $\Delta' \vdash_R \text{ob}(o', a', \text{idle}, \{p\}) \mathbf{ok}$. For NEW-COG-OBJECT, let $\Delta' = \Delta[o' \mapsto C, c' \mapsto \text{cog}]$. Since *fresh*(c'), this does not affect the well-typedness of o and o' . We must additionally show that $\Delta' \vdash_R \text{cog}(c', o') \mathbf{ok}$, which is immediate.
- **ASYNC CALL.** Let $\Delta \vdash_R \text{ob}(o, a, \{l|x = e!m(\overline{e}); q\}) \mathbf{ok}$. We first consider the case $e \neq \mathbf{this}$. By ASYNC CALL, we may assume that $\Delta \vdash e!m(\overline{e}) : \text{fut}\langle T \rangle$ and by ASSIGN that $\Delta(x) = \text{fut}\langle T \rangle$. Therefore, $\Delta \vdash e : I$ and $\Delta \vdash \overline{e} : \overline{T}$ such that *match*($m, \overline{T} \rightarrow T, I$). Assume that $\llbracket e \rrbracket_{aol} = o'$ and let $\Delta(o') = C$ for some

class C . By Lemma [11](#) there is a Δ' such that $\Delta' \vdash_R \llbracket e \rrbracket_{aol} : I$ and $\Delta'(o') = C$, so $C \preceq I$. By assumption class definitions are well-typed, so for any class C that implements interface I we have $match(m, \bar{T} \rightarrow T, C)$. By Lemma [11](#), $\llbracket \bar{e} \rrbracket_{aol}$ similarly preserves the type of \bar{e} . Let $\Delta'' = \Delta'[f \mapsto \text{fut}\langle T \rangle]$. Since $fresh(f)$ we know that $f \notin \text{dom}(\Delta')$, so if $\Delta' \vdash_R \text{cn } \mathbf{ok}$, then $\Delta'' \vdash_R \text{cn } \mathbf{ok}$. Since $\Delta' \vdash e!m(\bar{e}) = \Delta''(f)$, we get $\Delta'' \vdash_R \text{ob}(o, a, \{l|x = f; s\}, q) \mathbf{ok}$. Furthermore, $\Delta' \vdash \text{invoc}(o', f, m, \bar{v}) \mathbf{ok}$ and $\Delta'' \vdash_R \text{fut}(f, \perp) \mathbf{ok}$. The case $e = \mathbf{this}$ is similar, but uses the class of \mathbf{this} directly for the match (so internal methods are also visible).

- BIND-MTD. Let $C = \Delta(o)$. By assumption $\Delta \vdash_R \text{invoc}(o, f, m, \bar{v}) \mathbf{ok}$ and $\Delta \vdash_R \text{ob}(o, a, p, q) \mathbf{ok}$, so $\Delta(f) = \text{fut}\langle T \rangle$, $\Delta(\bar{v}) = \bar{T}$, and $match(m, \bar{T} \rightarrow T, C)$. Let \bar{x} be the formal parameters of m in C . Consequently, the auxiliary function $bind(o, f, m, \bar{v}, C)$ returns a process $\{l[\bar{T} \ \bar{x} \ \bar{v}, \text{fut}\langle T \rangle \ \text{destiny } f] | s\}$ which is well-typed in $\Delta \circ \text{fields}(C)$, and it follows that $\Delta \vdash_R \text{ob}(o, a, p, q \cup \{bind(o, f, m, \bar{v}, C)\}) \mathbf{ok}$.
- RETURN. By assumption, we have $\Delta \vdash_R \text{ob}(o, a, \{l|\mathbf{return } e; s\}, q) \mathbf{ok}$ and $\Delta \vdash_R \text{fut}(f, \perp) \mathbf{ok}$. Obviously, $\Delta \vdash_R \text{ob}(o, a, \{l|s\}, q) \mathbf{ok}$. Since $l(\text{destiny}) = f$ and l is well-typed, we know that $\Delta(\text{destiny}) = \Delta(f)$. Let $\Delta(f) = \text{fut}\langle T \rangle$. By T-RETURN, $\Delta \vdash_R e : T$ and by Lemma [11](#), $\Delta(v) = T$, so $\Delta \vdash_R \text{fut}(f, v) \mathbf{ok}$.
- READ-FUT. By assumption, $\Delta \vdash_R \text{ob}(o, a, \{l|x = e.\mathbf{get}; s\}, q) \mathbf{ok}$, $\Delta \vdash_R \text{fut}(f, v) \mathbf{ok}$, and $\llbracket e \rrbracket_{aol} = f$. Let $\Delta(f) = \text{fut}\langle T \rangle$. Consequently, $\Delta \vdash_R e.\mathbf{get} : T$ and $\Delta(v) = T$, so $\Delta \vdash x = v$, and $\Delta \vdash_R \text{ob}(o, a, \{l|x = v; s\}, q) \mathbf{ok}$.
- REM-SYNC-CALL. By assumption, $\Delta \vdash_R \text{ob}(o, a, \{l|x = e.m(\bar{e}); s\}, q) \mathbf{ok}$, $\Delta \vdash e.m(\bar{e}) : T$, and $fresh(f)$. Let $\Delta' = \Delta[f \mapsto \text{fut}\langle T \rangle]$. Then obviously $\Delta' \vdash f = e!m(\bar{e}); x = f.\mathbf{get}$.
- SELF-SYNC-CALL and COG-SYNC-CALL. By assumption, the judgments $\Delta \vdash_R \text{ob}(o, a, \{l|x = e.m(\bar{e}); s\}, q) \mathbf{ok}$, $\Delta \vdash e.m(\bar{e}) : T$, $\Delta \vdash_R \{l'|s'\} \mathbf{ok}$, and $fresh(f)$ hold. Let $\Delta' = \Delta[f \mapsto \text{fut}\langle T \rangle]$. Obviously $\Delta' \vdash \{l'|s'\}; \mathbf{cont}(f) \mathbf{ok}$, $\Delta' \vdash x = f.\mathbf{get}$, and $\Delta' \vdash_R \text{fut}(f, \perp) \mathbf{ok}$. \square

7 Tool Support

The ABS language is being used and developed as part of the EU project HATS (www.hats-project.eu). ABS comes with considerable tool support, including editing, compiling, running, and visualizing ABS models in the Emacs editor and in the Eclipse integrated development environment.

Compiler frontend. All ABS tools use a common compiler frontend which supplies parsing, type checking, and basic error reporting. The compiler frontend is implemented using the JastAdd toolkit [\[19\]](#) and provides an object-oriented, type-annotated syntax tree representing an ABS model. All backend implementations, code analyzers, etc. are implemented on top of this common base. At present there are two language backends, making ABS executable on the Maude rewriting engine and the Java virtual machine, with more backends planned.

The Maude backend is a translation of the operational semantics given in this paper into equational logic for the functional level of ABS and rewriting logic for

the concurrent object level. This semantics is executed as a language interpreter using the Maude tool [10]. Compiling an ABS model into Maude results in a set of class and function definitions (since all type checking is done at compile time, interface and datatype declarations do not have runtime representations). A special class implements the *main* block; starting an ABS model in Maude means instantiating an object of that class. The conciseness and high level of abstraction of the Maude backend make it well-suited for experiments with language constructs and semantics. Maude also provides model-checking support, but the large size of each state, as well as the non-deterministic scheduling and concurrent execution of ABS, and the resulting combinatorial explosion, make model-checking ABS models of realistic size very difficult in practice.

The *Java backend* provides a translation of ABS models into Java source code, which is compiled into bytecode using the standard Java tool chain. The Java backend uses a Java translation similar to the one for JCoBox [34], which proved to be very efficient. Compared to JCoBox, the generated code of ABS has better support for configuring the scheduling strategies, for system observation, and debugging. The ABS *main* block is translated into a standard Java main method so the generated code can be executed like standard Java programs.

The Java backend provides higher execution speed, an integration into existing Java tools, and the potential for integrating “native” or library functionality (e.g., file handling) into the language. Hence, the Java and Maude backends provide complementary and attractive features for the modeler.

8 Related Work

The concurrency model provided by concurrent objects and in actor-based computation, where software units with encapsulated processors communicate asynchronously, is increasingly attracting attention due to its intuitive and compositional nature (e.g., [2, 5, 9, 13, 21, 37]). ABS uses the communication mechanisms of Creol [24] for remote communication, based on asynchronous method calls and first-class futures [13]. A distinguishing feature of Creol is the cooperative scheduling between asynchronously called methods [24], which allows active and reactive behavior to be combined within objects as well as compositional verification of partial correctness properties [3, 13]. Creol’s model of cooperative scheduling has recently been generalized from single objects to groups of objects in a Java extension called JCoBox [34], which forms the basis for cogs in ABS.

Formal models are useful to clarify the intricacies of object orientation and may contribute to improve programming languages by making programs easier to understand, maintain, and analyze. Object calculi such as the ζ -calculus [1] and its concurrent extension [20] aim at directly expressing features such as self-reference, encapsulation, and method calls, but asynchronous method calls are not addressed. This also applies to Obliq [8], a programming language using similar primitives to target distributed concurrent objects. The object calculus of Di Blasio and Fisher [16] has both synchronous and asynchronous method calls, but, in contrast to ABS, return values are discarded when methods are

called asynchronously and the synchronous and asynchronous calls have different semantics. Caromel, Henrio, and Serpette propose ASP [9], a concurrent object calculus with asynchronous method calls and first-class futures. Compared to ABS, ASP’s futures are transparent (i.e., there is no polling and the get-operation is implicit) and communication is ordered to make reductions deterministic.

The internal concurrency model of cogs in ABS follows Creol’s concept of *cooperative scheduling* [24], but is lifted from the level of objects to the level of cogs. Synchronous method calls inside a cog are reentrant, which allows standard recursive programming of internal imperative data structures. Cogs in ABS may be compared to monitors [22] or to thread pools executing on a single processor. In contrast to monitors, explicit signaling is avoided. Sufficient signaling is ensured by the semantics, which significantly simplifies reasoning [12]. However, general monitors may be encoded in the language [24]. In contrast to thread pools, processor release is explicit. The activation of suspended processes is non-deterministically handled by an unspecified scheduler. Consequently, intra-object concurrency is similar to the interleaving semantics of concurrent process languages [4, 17], and each process resembles a series of guarded atomic actions (ignoring local variable scopes). Internal reasoning control is facilitated by the explicit declaration of release points, at which class invariants should hold [3, 18].

Our type system resembles that of Featherweight Java [23], a core calculus for Java, because of its nominal approach. Featherweight Java is class-based with single inheritance, and subtyping is the reflexive and transitive closure of the subclass relation. In contrast, ABS cleanly distinguishes classes and types. Creol combined asynchronous calls and interfaces as in ABS with class inheritance, choice operators, and a notion of *cointerface* at the interface level to accommodate type-safe callbacks [25]. Creol’s type system used an effect system [28] to infer types for future variables, which allowed a flexible reuse of future variables for method calls with different return types. By means of backwards analysis, the effect system could insert deallocation operations to garbage-collect inaccessible futures depending on the local control flow at runtime [26]. In contrast, future variables in ABS have explicit types for return values, which restricts reuse but allows a type analysis without effects. Compared to previous work on Creol, this paper formalizes user-defined data types and functions in the context of concurrent objects. The presented type safety results show how the typing context is dynamically extended when new objects and futures are created.

9 Conclusion

This paper presents ABS, an abstract behavioral specification language for designing executable, object-oriented, formal models of distributed systems. The language is situated between design-oriented, foundational, and implementation-oriented languages by being abstract, yet executable. The concurrency model of ABS is based on concurrent object groups (cogs) which are encapsulated behind interfaces and do not share state. While cogs may execute in parallel, there is a cooperative model of interleaving concurrency inside each cog, reflected by explicit processor release points in the language. This concurrency model is

inherently compositional and allows to reason about concurrent system behavior using monitor invariants and sequential object-oriented proof systems.

The combination of a functional and a concurrent object level in the ABS language allows the modeler to focus the model on crucial parts of an imperative system, including its concurrency and synchronization mechanisms, by using functional data types to abstract from other parts of the internal data structures and by abstracting from specific scheduling policies and environmental properties. ABS is a formally defined, executable specification language. We gave rigorous, mathematical definitions of its core syntax, type system, and operational semantics. We proved a subject reduction result showing that execution preserves well-typedness in the sense that “method not understood” errors do not occur for well-typed ABS models.

Acknowledgment. We thank Frank S. de Boer, Olaf Owe, and the HATS consortium for interesting discussions and their contributions to the ABS, and the anonymous referees for excellent feedback, significantly improving the paper.

References

1. Abadi, M., Cardelli, L.: *A Theory of Objects*. Springer, Heidelberg (1996)
2. Agha, G.A.: *ACTORS: A Model of Concurrent Computations in Distributed Systems*. The MIT Press, Cambridge (1986)
3. Ahrendt, W., Dylla, M.: A system for compositional verification of asynchronous objects. *Science of Computer Programming* (2010) (In press)
4. Andrews, G.R.: *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison-Wesley, Reading (2000)
5. Armstrong, J.: *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf (2007)
6. Barnett, M., Leino, K.R.M., Schulte, W.: The spec# programming system: An overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) *CASSIS 2004. LNCS*, vol. 3362, pp. 49–69. Springer, Heidelberg (2005)
7. Burdy, L., Cheon, Y., Cok, D.R., Ernst, M.D., Kiniry, J.R., Leavens, G.T., Leino, K.R.M., Poll, E.: An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer (STTT)* 7(3) (June 2004)
8. Cardelli, L.: A language with distributed scope. *Comp. Sys.* 8(1), 27–59 (1995)
9. Caromel, D., Henrio, L., Serpette, B.P.: Asynchronous sequential processes. *Information and Computation* 207(4), 459–495 (2009)
10. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.L. (eds.): *All About Maude - A High-Performance Logical Framework*. LNCS, vol. 4350. Springer, Heidelberg (2007)
11. Clements, P.C.: A survey of architecture description languages. In: *Proc. Workshop on Software Specification and Design (IWSSD 1996)*, pp. 16–25. IEEE, Los Alamitos (1996)
12. Dahl, O.-J.: Monitors revisited. In: *A Classical Mind, Essays in Honour of C.A.R. Hoare*, pp. 93–103. Prentice Hall, Englewood Cliffs (1994)
13. de Boer, F.S., Clarke, D., Johnsen, E.B.: A complete guide to the future. In: De Nicola, R. (ed.) *ESOP 2007. LNCS*, vol. 4421, pp. 316–330. Springer, Heidelberg (2007)
14. Full ABS Modeling Framework, Deliverable 1.2 of project FP7-231620 (HATS) (March 2011), <http://www.hats-project.eu>

15. Verification of Behavioral Properties, Deliverable 2.5 of project FP7-231620 (HATS) (March 2011), <http://www.hats-project.eu>
16. Di Blasio, P., Fisher, K.: A calculus for concurrent objects. In: Sassone, V., Montanari, U. (eds.) CONCUR 1996. LNCS, vol. 1119, pp. 655–670. Springer, Heidelberg (1996)
17. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM* 18(8), 453–457 (1975)
18. Dovland, J., Johnsen, E.B., Owe, O.: Observable behavior of dynamic systems: Component reasoning for concurrent objects. In: Proc. Foundations of Interactive Computation (FInCo 2007). ENTCS, vol. 203, pp. 19–34. Elsevier, Amsterdam (2008)
19. Ekman, T., Hedin, G.: The JastAdd system: modular extensible compiler construction. *Science of Computer Programming* 69(1-3), 14–26 (2007)
20. Gordon, A.D., Hankin, P.D.: A concurrent object calculus: Reduction and typing. In: Proc. High-Level Concurrent Languages (HLCL). ENTCS, vol. 16(3) (1998)
21. Haller, P., Odersky, M.: Scala actors: Unifying thread-based and event-based programming. *Theor. Comp. Sci.* 410(2-3), 202–220 (2009)
22. Hoare, C.A.R.: Monitors: an operating systems structuring concept. *Communications of the ACM* 17(10), 549–557 (1974)
23. Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Prog. Lang. and Sys* 23(3), 396–450 (2001)
24. Johnsen, E.B., Owe, O.: An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling* 6(1), 35–58 (2007)
25. Johnsen, E.B., Owe, O., Yu, I.C.: Creol: A type-safe object-oriented model for distributed concurrent systems. *Theor. Comp. Sci.* 365(1-2), 23–66 (2006)
26. Johnsen, E.B., Yu, I.C.: Backwards type analysis of asynchronous method calls. *Journal of Logic and Algebraic Programming* 77, 40–59 (2008)
27. Larsen, K.G., Pettersson, P., Yi, W.: Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer (STTT)* 1(1-2), 134–152 (1997)
28. Lucassen, J.M., Gifford, D.K.: Polymorphic effect systems. In: Proc. POPL, pp. 47–57. ACM Press, New York (1988)
29. Magee, J., Dulay, N., Eisenbach, S., Kramer, J.: Specifying distributed software architectures. In: Botella, P., Schäfer, W. (eds.) ESEC 1995. LNCS, vol. 989, pp. 137–153. Springer, Heidelberg (1995)
30. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. *Theor. Comp. Sci.* 96, 73–155 (1992)
31. Milner, R.: *Communicating and Mobile Systems: the π -Calculus*. Cambridge University Press, Cambridge (1999)
32. Pierce, B.C.: *Types and Programming Languages*. The MIT Press, Cambridge (2002)
33. Plotkin, G.D.: A structural approach to operational semantics. *Journal of Logic and Algebraic Programming* 60-61, 17–139 (2004)
34. Schäfer, J., Poetzsch-Heffter, A.: JCoBox: Generalizing active objects to concurrent components. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 275–299. Springer, Heidelberg (2010)
35. van Deursen, A., Klint, P.: Domain-specific language design requires feature descriptions. *Journal of Computing and Information Technology* 10(1), 1–18 (2002)
36. Warmer, J., Kleppe, A.: *The Object Constraint Language: Precise Modelling with UML*. Object Technology Series. Addison-Wesley, Reading (1999)
37. Welc, A., Jagannathan, S., Hosking, A.: Safe futures for Java. In: Proc. OOPSLA, pp. 439–453. ACM, New York (2005)

A Component Model for the ABS Language^{*}

Michaël Lienhardt¹, Ivan Lanese¹, Mario Bravetti¹, Davide Sangiorgi¹,
Gianluigi Zavattaro¹, Yannick Welsch²,
Jan Schäfer², and Arnd Poetzsch-Heffter²

¹ Focus Team, University of Bologna, Italy

{`lienhard,lanese,bravetti,davide.sangiorgi,zavattar`}@cs.unibo.it

² Software Technology Group, University of Kaiserslautern, Germany
{`welsch,jschaefer,poetzsch`}@cs.uni-kl.de

Abstract. Finding good abstractions to model and express *partial update*, *mobility* and *wrapping* in object-oriented systems remains challenging. In this paper, we propose COMP, a process calculus approach for component models that merges aspects of object-orientation and *evolution*. The key features of COMP are: a hierarchical structure of components; the capacity to move, update, wrap components; method interfaces for components; and some isolation capacities to encode distribution and wrapping. Specifically, we introduce the syntax of COMP and formulate its operational semantics. We show a number of examples of use of COMP, with particular emphasis on common evolution patterns for components.

1 Introduction

Evolution is an important issue in complex software systems. The needs and requirements on a system may change over time. This may happen because the original specification was incomplete or ambiguous, or because new needs arise that had not been predicted at design time. As designing and deploying a system is costly, it is important that the system supports operations to adapt it to changes in the surrounding environment. By *dynamic evolution* of a system we refer to the possibility that functionalities offered by the system change over time. This may involve reconfiguring and updating applications to meet new requirements and new operating conditions, which were unexpected when the application was developed and deployed.

The goal of this work is to isolate interesting constructs for expressing dynamic evolution that can be easily integrated into existing object-oriented programming and modeling languages. More specifically, we are interested in integrating these constructs into the modeling language ABS [10], a language developed within the HATS European project [12], which is based on ideas from Creol [13,9] and JCoBox [27]. The ABS language is a formally defined object-oriented language whose purpose is to support the design of concurrent and distributed software systems, in particular by providing high-level specification and checking facilities. To integrate well with ABS (and similar object-oriented languages), we

^{*} Partly funded by the EU project FP7-231620 HATS.

adopt its basic scheme to handle components¹. Components are represented by objects. The object's methods enable communication between the object and its environment. Furthermore, the same language specification technique is used to foster the integration between the calculus and the formal analysis techniques developed for ABS and Creol [9,1].

Many different component models were developed over the past decade, like OSGI [2], FRACTAL [4], COM [8], JAVA BEANS [30] and others [3,21,23]. These models focus on aspects different from those of COMP (see discussion in Sect. 5). If at all possible, dynamic evolution steps have to be hardcoded in these models (e.g. in OSGI, components can be stopped and replaced by new versions, but this has to be programmed using the framework API). In particular, a high-level formal analysis of such steps is not supported. The central aspects and goals of COMP are summarized in the following.

Firstly, these previous models [2,4,8,30] are not developed for formal analysis and do not provide a formal semantics needed for verification. In particular, they are more or less coupled to complex programming languages and APIs written in these languages which makes it difficult to identify an analyzable formal core.

The second aspect is about interface specifications based on typing and ports. Many component models (e.g. [4,18,23]) support typed input and output ports to allow static checking of composition. To focus on dynamic evolution and keep the calculus manageable, we do not consider typing aspects in the core calculus presented in this paper. Similar to objects, a component has an interface consisting of a set of (untyped) methods. The component can communicate via channels with other components. In Sect. 4 we show how the basic constructs of the core calculus can be used to model more structured connections between components. An extension to static analysis techniques (e.g., type systems) is planned as future work.

The third aspect concerns scope and component visibility. In many component models [4,22,29], the hierarchical structure of a component system is rigid: the boundary of a component a hides all the inner components, which are unreachable from a 's environment. Other component models do not support nesting of components at all. COMP provides component nesting and flexibility with respect to visibility of internal components in order to model common patterns of distributed systems that involve sharing of resources.

The last issue is about the *passivation* mechanism provided by some of the models such as the KELL-calculus [29] or MECO [22]. Passivation allows the programmer to freeze a component and capture it; the component can then be sent around at will, with even the possibility of duplicating it. Passivation is very powerful, but makes it quite hard to ensure safety of reconfiguration and to prove properties of systems. Also, the practical relevance of the act of copying a running component is dubious.

The language that we propose in this paper, called COMP, addresses the four issues above as follows:

¹ As we focus on dynamic aspects, components here are runtime entities, often called component instances to distinguish them from the programs.

- COMP has a formal semantics, defined using the reduction and labeled transition system styles.
- Components in COMP have an input interface, but no output interfaces; as a consequence, components can be used much in the same way as objects; however, component additionally have mobility capacities.
- COMP provides opening and closing operations for dynamically changing the visibility of a component. Thus, while by default communication in COMP remains global to fit the communication capacities of objects, if needed, the boundary of a component a can be closed to restrict access to specific components internal to a , say b ; as a consequence, a component external to a will not be able to directly access b any more.
- Mobility of running components is allowed by means of movement primitives rather than by capturing and communicating components. These primitives are inspired by the constructs for achieving mobility in the Ambient calculus [6]. In COMP a component may thus move in the tree hierarchy of a component system. Processes (including component definitions) may be communicated, but they cannot be grabbed when running.

The main challenge in the formalization of a component model is to isolate key aspects of component-based systems and reflect them into specific constructs. In our case, the scenarios we want to describe with our model are presented as a set of examples that show how objects, components and runtime modifications of a program architecture can easily be combined. While developing the model we followed a general strategy that we call “*the architect principle*”, which means that each component in the system is in charge of managing its children. We chose to follow this principle in our model because it improves its consistency, making it more easy to work with. Indeed, because of this principle, the one responsible for a modification is uniquely identified, which makes the behavior of systems clearer, and the modifications easier to code. Moreover, as the parent has the control over its children and their communications, it can help their integration into the rest of the system. We summarize below the main concepts that our approach is based on.

Components. Components are a way to structure a system into a set of units, each of them having a clear boundary. Processes inside the same boundary share some features, which can be of various kinds. For instance, they may share some computational resources, they may be in the same physical location or just inside the same security perimeter, or they may jointly implement some functionalities. In our case each component has its own data and its own execution space. More important, components represent the units of evolution in this paper. Put differently, evolution is obtained by adding, removing, replacing, wrapping or manipulating components and the component structure in a modular way. Notably, while a component is being, e.g., replaced, other components can continue to execute normally, thus minimizing service disruption.

Components give rise to a hierarchical structure, which allows for the modular definition of complex software architectures. Nesting of components may have

different meanings, corresponding to the meanings of components themselves. In our case the parent of a component is in charge of updating its children, according to the “architect principle” described above.

Components can also be used to specify system deployment. Components in fact may represent physical locations and available resources, and deployment can be done by specifying how to associate to each object its enclosing component, thus defining how to deploy it. Notably, evolution constructs discussed later enable dynamic re-deployment of components to deal with modifications of the underlying architecture. We will not consider this possibility in depth.

Methods. Each component is equipped with a set of methods, defining its available functionalities. Having an explicit *input interface* is important, since this provides an abstract description of the functionalities of the component, to be used to ensure correctness of evolution steps. For instance, if an evolution step preserves the interface of the involved components, it will introduce no typing errors on runtime invocations.

Having methods in a component also matches the intuition that components correspond to objects with extended capabilities, which is useful for integration of components with objects. In contrast to other component models, COMP components have no output interface, since this has no correspondence inside objects. This would also make the semantics of the isolation mechanism more complex.

Isolation. A consequence of the component hierarchy is that a component may decide to hide its internal components, or to make (some of) them available to the external world. Hiding is fundamental for encapsulation: hidden components cannot be reached directly from the outside. Isolation can also be used to encode wrapping, where the wrapper component hides the wrapped one, while providing updated functionalities. In this way, for instance, methods can be removed, added or redefined. An internal component may however be left visible, which is useful for modeling shared resources.

According to the “architect principle”, the decision about when and whether making a component available to the outside is taken by its parent.

Mobility. Having a hierarchical structure in place, the immediate way for achieving evolution is to allow components to move along the hierarchy. Remember that mobility can be either logical or physical, according to whether components model the software architecture of the system or its physical distribution.

Clearly, different primitives for mobility are possible. We decided to introduce two primitives for mobility, **in** and **out**, inspired from the Ambient calculus [6], that allow us to move a component inside a sibling one or outside its parent. These primitives disallow direct mobility between locations far from each other (unrealistic in many cases, e.g., for physical locations). Again, following the architect principle, a component may only be moved by its parent.

Channel-based communication. Our components communicate using channels. Analogously to passing object references, this kind of communication is

completely independent from the component hierarchy. The main goal of this mechanism is to enable the support of futures (as available in ABS) for returning the result of a method to the caller regardless of which reconfigurations occurred since the call has been performed. Also, channel-based communication can be handy to synchronize different components performing some joint reconfiguration. We allow both names and processes to be communicated. Process communication in particular allows for code injection. Code injection and mobility are quite different concepts, since mobility refers to running components while code injection concerns idle processes.

Structure of the paper. We first present the calculus and explain its syntax in Sect. 2. We then formulate its operational semantics in Sect. 3, which is introduced by both: i) a reduction semantics that models the manipulation of the component's structure; and ii) a labeled semantics that models method invocations and channel-based communication. We illustrate the calculus with a number of examples in Sect. 4 showing, in particular, how various patterns of evolution of components are captured. Finally, we discuss related work in Sect. 5 and conclude in Sect. 6.

2 Primitives for Components and Evolution

This section presents the primitives we propose for modeling components and their evolution patterns. The formal semantics will be discussed in the next section. The syntax of COMP primitives is summarized in Fig. 1. Our syntax is based on a few syntactic categories (pairwise distinct): names for components, ranged over by a, b, l, x , names for channels, ranged over by c, d, y, ack, ch , names for methods, ranged over by m , and process variables, ranged over by X . We denote with S isolation sets, namely sets of component names that can be of two kinds: finite sets, represented as $\{a_1, \dots, a_n\}$ ($\{a\}$ will be shortened into a), or sets including all the component names but a finite set, represented as $\overline{\{a_1, \dots, a_n\}}$.

Our values include component names, channel names and processes, and we use V to range over values. We use n to range over both component and channel names. We use J to range over process variables and component and channel names (assuming in this last case that they are bound by an input or a method definition).

The main construct of COMP is the component $a(S)\{M\}[P]$, where i) a is the name of the component; ii) S is the isolation set containing the names of inner components that are hidden from the environment; iii) M is the set of methods of the component; and iv) P is the body of the component, containing its currently running code and its sub-components.

Methods are defined as usual: $m(J).P$ declares a method m with formal parameter J and body P . As already said, J can be a component name, a channel name, or a process variable.

² We consider just one parameter, the extension to many parameters is trivial.

<i>Component Names</i>	a, b, x	
<i>Channel Names</i>	c, d, y, ack, ch	
<i>Process Variables</i>	X	
<i>Names</i>	$n ::= a \mid c$	
<i>Values</i>	$V ::= P \mid n$	
<i>Placeholders</i>	$J ::= X \mid n$	
<i>Isolation Sets</i>	$S ::= \{a_1, \dots, a_n\} \mid \overline{\{a_1, \dots, a_n\}}$	
<i>Processes</i>	$P ::= a(S)\{M\}[P]$	Component
	$\mid P \mid P$	Parallel Composition
	$\mid A.P$	Action Prefix
	$\mid 0$	Null Process
	$\mid X$	Process Variable Occurrence
	$\mid \nu n P$	Restriction
<i>Actions</i>	$A ::= a.m(V)$	Method Call
	$\mid \mathbf{open} S$	Open
	$\mid \mathbf{close} S$	Close
	$\mid a \mathbf{in} b$	Move In
	$\mid a \mathbf{out} b$	Move Out
	$\mid c(J)$	Message Receive
	$\mid c(V)$	Message Send
<i>Methods Sets</i>	$M ::= 0$	Empty Methods Set
	$\mid m(J).P$	Method Definition
	$\mid M \mid M$	Methods Set

Fig. 1. COMP syntax

Processes can perform actions. The main actions are: i) method invocation $a.m(V)$, that calls the method m of the component a with parameter V (here V is either a process P , for code injection, a component name b , or a channel name c); ii) **close** S , that hides the child components in the isolation set S from the rest of the system; in particular, **close** $\bar{\emptyset}$ will make the component a perfect black-box w.r.t. method invocations; iii) **open** S , that, on the opposite, reveals the components in S to the environment; iv) $a \mathbf{in} b$, that puts the component a inside the parallel component b and v) $a \mathbf{out} b$, that takes the component a that is inside b , and puts it outside as shown in Fig. 2. The command $a(S)\{M\}[P]$ creates a new component named a .

Finally, COMP contains several other constructs, standard from process calculi such as π -calculus [20] and Higher-Order π -calculus [26]: i) 0 is the process with no behavior (like **skip** in imperative languages); ii) X is a process variable; iii) $\nu n P$ is (channel or component) name restriction, which creates a new name n ; and iv) $c(J)$, $c(V)$ are input and output primitives for communication on channels.

As mentioned above, we use channel-based communication, well studied in process calculi, mainly to model the **return** statement of object-oriented languages. Remember that ABS has asynchronous method invocations and the

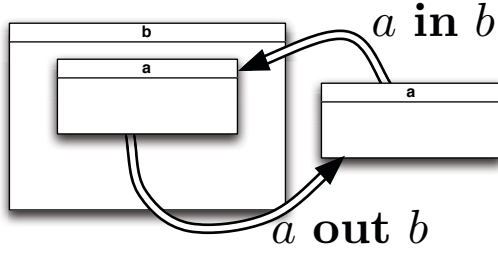


Fig. 2. In and Out primitives

communication of the return value is done using futures [9]. As one can see in the evolution patterns presented in Sect. 4, most of the occurrences of channel-based communication are already used for returning values.

COMP has no explicit operator for recursion or replication. However infinite behaviors can be encoded using higher-order communication or self-involutions of methods.

Bound names and variables (binders are name restriction, method definitions and input on channels) can be α -converted as usual. We write $n(P)$ for the set of names in process P and $fn(P)$ for its set of free names. We restrict our attention to processes with no free variables. We remark that names in isolation set $\{a_1, \dots, a_n\}$ are only a_1, \dots, a_n .

3 Operational Semantics

The operational behavior of a process calculus is usually defined either by means of a reduction semantics, or by means of a labeled transition semantics. A reduction semantics typically uses the auxiliary relation of structural congruence, with which the participants of an interaction are brought into contiguous positions. This makes it possible to express interaction by means of simple term-rewriting rules. In a labeled transition semantics, in contrast, interacting parties can be far away, and the labels of the transition carry the information on the interaction up in the term, allowing synchronization.

For our calculus, we express component reconfigurations (those derived from mobility of components and modification of the isolation sets) by means of a reduction semantics, whereas channel-based interactions and method calls are described by means of a labeled transition system (LTS). The reason for the separation is that component reconfiguration is a local activity, whereas channel/method interaction is global (i.e., components far away can interact). The semantics of reconfiguration is simpler via a reduction semantics in the same way as in the Ambient calculus [6], which has inspired our movement primitives, and has a simple reduction semantics but a complex LTS semantics [19].

We write $P \rightarrow P'$ for an execution step of the process P that is derived using the reduction semantics, and $P \xrightarrow{\mu} P'$ for a step derived using the labeled semantics in which the label is μ . Finally, \triangleright is the union of the two relations \rightarrow and $\xrightarrow{\mu}$, and \triangleright^*

is the reflexive and transitive closure of \triangleright . The relations \rightarrow and $\xrightarrow{\mu}$ are presented in the following two sections.

3.1 Semantics of Reconfiguration

We discuss here the formal semantics of reconfiguration in COMP. As already said, this consists in defining a structural congruence relation and a reduction relation \rightarrow .

Structural congruence. The structural congruence relation is written \equiv , and is defined as the smallest congruence that satisfies the rules presented in Fig. 3. The structural congruence \equiv is quite standard: the parallel operator is commutative and associative and has unit 0 for processes and methods, while name restriction can be extruded if not capturing free names.

$$\begin{array}{l}
 P \mid 0 \equiv P \qquad P_1 \mid P_2 \equiv P_2 \mid P_1 \qquad (P_1 \mid P_2) \mid P_3 \equiv P_1 \mid (P_2 \mid P_3) \\
 M \mid 0 \equiv M \qquad M_1 \mid M_2 \equiv M_2 \mid M_1 \qquad (M_1 \mid M_2) \mid M_3 \equiv M_1 \mid (M_2 \mid M_3) \\
 Q \mid \nu n P \equiv \nu n (Q \mid P) \quad \text{if } n \notin \text{fn}(Q) \\
 a(S)\{M\}[\nu n P] \equiv \nu n (a(S)\{M\}[P]) \quad \text{if } n \notin \text{fn}(S) \cup \{a\} \cup \text{fn}(M)
 \end{array}$$

Fig. 3. Structural congruence on COMP processes

Reduction rules. The relation \rightarrow is defined as the smallest relation closed w.r.t. \equiv and α -conversion that validates the rules presented in Fig. 4. The first two rules describe the manipulation of the program architecture. First, the command a in b takes two parallel components named a and b respectively and moves a

$$\begin{array}{l}
 a \text{ in } b.P \mid a(S_a)\{M_a\}[P_a] \mid b(S_b)\{M_b\}[P_b] \rightarrow P \mid b(S_b)\{M_b\}[P_b] \mid a(S_a)\{M_a\}[P_a] \\
 a \text{ out } b.P \mid b(S_b)\{M_b\}[P_b] \mid a(S_a)\{M_a\}[P_a] \rightarrow P \mid a(S_a)\{M_a\}[P_a] \mid b(S_b)\{M_b\}[P_b] \\
 a(S)\{M\}[\text{close } S'.P \mid P'] \rightarrow a(S \cup S')\{M\}[P \mid P'] \\
 a(S)\{M\}[\text{open } S'.P \mid P'] \rightarrow a(S \setminus S')\{M\}[P \mid P'] \\
 \frac{P \rightarrow P'}{P \mid Q \rightarrow P' \mid Q} \qquad \frac{P \rightarrow P'}{\nu n P \rightarrow \nu n P'} \qquad \frac{P \rightarrow P'}{a(S)\{M\}[P] \rightarrow a(S)\{M\}[P']}
 \end{array}$$

Fig. 4. Rules for reconfiguration

inside b . The operator a **out** b has the opposite behavior, as it takes a component named b in parallel (i.e. in the same parent component), takes a component a inside b and puts a outside of it (i.e. in parallel to b). The two following rules allow to change component visibility. The action **close** S executed inside a component $a(S')\{M\}[P]$ will add S to S' while **open** S will remove S from S' . Since isolation sets are closed under set union and set difference the semantics of the two primitives is well-defined.

The last three rules are standard closures under parallel composition, name restriction and component contexts.

How the isolation set S influences the behavior of method calls is described in the following section.

3.2 Method Invocations and Channel Communications

Method invocations and channel communications are described using an LTS. The rules for channel communication are entirely standard, following the SOS style of message-passing calculi such as π -calculus and Higher-Order π -calculus, as channel-based communications are independent from the component hierarchy. Method calls are described by similar rules, which just have one more side condition that checks whether the method call is allowed to go through the component boundaries it has to cross.

The label (or action) μ of a transition can be of five different kinds: i) τ , corresponding to an inner step; ii) $\bar{c}(V)$, corresponding to the sending of the value V on the channel c ; iii) $c(V)$, corresponding to the reception of the value V at the channel c ; iv) $\overline{a.m}(V)$, corresponding to the invocation of the method m of the component a with the parameter V ; and v) $a.m(V)$, corresponding to the triggering of method m of component a with the parameter V . To simplify our rules, we note Φ a label corresponding to a communication on a channel or the inner step τ , and $\Psi(a)$ a label corresponding to a method call to a component named a .

The relation $\xrightarrow{\mu}$ is defined as being the smallest closed relation w.r.t. \equiv and α -conversion that validates the rules in Fig. 5. The first two rules are the basic cases of channel communication, while the two following ones deal with the basic cases of method invocation. We remember that values V can be component or channel names or processes. Similarly, J is used for denoting either a (bound) component/channel name or a process variable. The five next rules handle congruence: i) first, we state that the relation is closed w.r.t. parallel composition; ii) then, communication on channels can freely cross component boundaries; while iii) method calls to a component b cannot cross a component boundary with b in its isolation set S ; also iv) an inner step is always propagated by components and v) can cross restrictions. We do not need to add rules for extrusions or propagating actions through restrictions, since restrictions can always be moved to the top level. These rules would be needed however, e.g., for defining a bisimilarity-based abstract semantics. The last rule shows how two processes synchronize to achieve either a channel communication or a method call (in labels, overline is only put on subjects for notational convenience), while the last one allows components to invoke their own methods.

$$\begin{array}{c}
c\langle V \rangle.P \xrightarrow{\bar{c}(V)} P \qquad c(J).P \xrightarrow{c(V)} P\{V/J\} \qquad a_{\mathbf{m}}\langle V \rangle.P' \xrightarrow{\overline{a_{\mathbf{m}}}(V)} P' \\
a(S)\{\mathbf{m}(J).P \mid M\}[P'] \xrightarrow{a_{\mathbf{m}}(V)} a(S)\{\mathbf{m}(J).P \mid M\}[P' \mid P\{V/J\}] \qquad \frac{P_1 \xrightarrow{\mu} P_2}{P_1 \mid P \xrightarrow{\mu} P_2 \mid P} \\
\frac{P_1 \xrightarrow{\Phi} P_2}{a(S)\{M\}[P_1] \xrightarrow{\Phi} a(S)\{M\}[P_2]} \qquad \frac{P_1 \xrightarrow{\Psi(b)} P_2 \quad b \notin S}{a(S)\{M\}[P_1] \xrightarrow{\Psi(b)} a(S)\{M\}[P_2]} \\
\frac{P_1 \xrightarrow{\tau} P_2}{\nu n P_1 \xrightarrow{\tau} \nu n P_2} \qquad \frac{P_1 \xrightarrow{\mu} P_2 \quad P_1' \xrightarrow{\bar{\mu}} P_2'}{P_1 \mid P_1' \xrightarrow{\tau} P_2 \mid P_2'} \\
\frac{P_1 \xrightarrow{\overline{a_{\mathbf{m}}}(V)} P_2}{a(S)\{\mathbf{m}(J).P \mid M\}[P_1] \xrightarrow{\tau} a(S)\{\mathbf{m}(J).P \mid M\}[P_2 \mid P\{V/J\}]}
\end{array}$$

Fig. 5. Rules for communication

4 Basic Reconfiguration Patterns

In this section we discuss different basic patterns of reconfiguration, showing how they can be encoded using COMP primitives.

Adding and removing a component. The two most basic operations to manipulate a program structure, in addition to the **in** and **out** operators, are the addition and removal of components. The addition of a component is straightforward, as it corresponds to the creation of a new component. The macro **Add**(a, S, M, P) creates a component with name a , isolation set S , set of methods M and body P :

$$\mathbf{Add}(a, S, M, P) \triangleq a(S)\{M\}[P]$$

The encoding of component removal is more subtle. Instead of destroying the component, we simply hide it with the insurance that it will never be accessible again. Macro **Remove**(a, P) removes component a and then executes continuation process P , that is supposed to notify the environment that the removal has been performed, i.e. a is not available anymore.

$$\mathbf{Remove}(a, P) \triangleq \nu b(b(\bar{0})\{0\}[0] \mid a \mathbf{in} b.P)$$

In order to remove a component, the **Remove**(a, P) process needs to be put in parallel with the component to be removed:

$$a(S)\{M\}[Q] \mid \mathbf{Remove}(a, P)$$

Assuming that the action a in b inside $\mathbf{Remove}(a, P)$ is executed, the configuration becomes

$$\nu b(b(\bar{\emptyset})\{0\}[a(S)\{M\}[Q]] \mid P)$$

The macro creates a new black-box component b that no one knows (and thus no one can modify), then moves the component a inside it and executes continuation P . Note that b completely hides a and its subcomponents from the rest of the architecture: a will never be accessible again. Hence, as soon as the computation inside a finishes, it becomes behaviorally equivalent to 0 and can be safely garbage collected.

Our definition of component removal intentionally does not destroy the component. Indeed, suppose we want to remove the component a so to replace it with a newer version. The current version of a could be computing some result of past method calls. Immediately destroying the component a would create some consistency issues inside the system, as some other computations may be waiting for the results being computed by a . Our definition ensures that we can replace a with another component without creating such inconsistencies.

Hiding structure manipulation. It is common practice to isolate the part of the system one wants to modify prior to its manipulation to prevent possible interferences. In our case, interferences may emerge because two components can have the same name: suppose we want to create a component b and move an existing component a inside of it. The natural approach is to first create the component b and then move a , but, if another component with name b already existed, we will have two components with the same name b but possibly different behavior/structure. Because of the nondeterminism, we cannot ensure that component a is moved in the newly created component. It may as well happen that it is moved inside the old one. A simple way to avoid this is to create a temporary component b' where we put only the components that are expected to take part in the reconfiguration, i.e. a and the new b . After the reconfiguration process terminates, we can put the resulting subsystem back in place.

We propose here a simple definition of this isolation mechanism. The macro $\mathbf{Hide}(a, b, P, ack, P')$ hides the component a , applies to it the reconfiguration specified by P which notifies its end by sending an output on ack . Upon receipt of ack the new component b is put in the environment. Finally, process P' is executed to make the environment aware of the end of the reconfiguration.

$$\mathbf{Hide}(a, b, P, ack, P') \triangleq \nu b'(b'(\bar{\emptyset})\{0\}[P] \mid a \text{ in } b'.ack(x).b \text{ out } b'.P')$$

An example of usage of the pattern will be discussed in the next paragraph about component renaming. In our macro definition, channel ack is used to notify when the reconfiguration specified by P is terminated and the resulting component b can be released to the environment. In order to avoid interferences this name should not be used elsewhere, e.g. it may be freshly generated just before the macro (a bit more care is needed if P is received via a communication).

Renaming a component. Since components are identified by their names, renaming a component is a useful operation. Also, two components may have the same name. This raises the possibility of nondeterminism in method invocations or reconfiguration operations – which might or might not be desired depending on the situation. Renaming a component can thus be used to create or remove nondeterminism.

We show below how to define macro **Rename**(a, b, P), which renames component a into b (we assume a and b to be distinct) and executes P to notify to the environment about the end of the renaming operation. The idea is to create a new component with name b , with isolation set $\{a\}$ and with the same methods as a (we assume to know the interface m_1, \dots, m_n of the component to be renamed, i.e. the macro depends on the interface of the component to be renamed). The behavior of these methods will be just to forward the calls to a . By moving a into b (thus making a no longer accessible from the environment), b behaves as a . This concludes the renaming. Remark that the renaming definition is based on the isolation macro **Hide** defined above to avoid interferences.

Rename(a, b, P)

$$\triangleq \nu ack \mathbf{Hide}(a, b, \left(b(a) \left\{ \begin{array}{l} m_1(x).a.m_1\langle x \rangle \\ \dots \\ m_n(x).a.m_n\langle x \rangle \end{array} \right\} [0] \mid a \mathbf{in} b.ack\langle 0 \rangle.0 \right), ack, P)$$

We show now how the renaming of a component works in practice. To simplify the presentation, we write M for the methods of the component a , and M' for the methods of b , defined as in the definition of the **Rename** operator.

$$\begin{aligned} & a(S)\{M\}[P] \mid \mathbf{Rename}(a, b, P') \\ &= a(S)\{M\}[P] \mid \nu ack \mathbf{Hide}(a, b, (b(a)\{M'\}[0] \mid a \mathbf{in} b.ack\langle 0 \rangle.0), ack, P) \\ &= a(S)\{M\}[P] \mid \nu ack, b' (b'(\bar{\emptyset})\{0\}[b(a)\{M'\}[0] \mid a \mathbf{in} b.ack\langle 0 \rangle.0] \mid \\ &\quad a \mathbf{in} b'.ack(x).b \mathbf{out} b'.P') \\ &\equiv \nu ack, b' (a(S)\{M\}[P] \mid b'(\bar{\emptyset})\{0\}[b(a)\{M'\}[0] \mid \\ &\quad a \mathbf{in} b.ack\langle 0 \rangle.0] \mid a \mathbf{in} b'.ack(x).b \mathbf{out} b'.P') \\ &\rightarrow \nu ack, b' (b'(\bar{\emptyset})\{0\}[a(S)\{M\}[P] \mid b(a)\{M'\}[0] \mid \\ &\quad a \mathbf{in} b.ack\langle 0 \rangle.0] \mid ack(x).b \mathbf{out} b'.P') \\ &\rightarrow \nu ack, b' (b'(\bar{\emptyset})\{0\}[b(a)\{M'\}[a(S)\{M\}[P]] \mid ack\langle 0 \rangle.0] \mid ack(x).b \mathbf{out} b'.P') \\ &\xrightarrow{\tau} \nu ack, b' (b'(\bar{\emptyset})\{0\}[b(a)\{M'\}[a(S)\{M\}[P]]] \mid b \mathbf{out} b'.P') \\ &\rightarrow \nu ack, b' (b'(\bar{\emptyset})\{0\}[0] \mid b(a)\{M'\}[a(S)\{M\}[P]] \mid P') \end{aligned}$$

Wrapping. Wrapping is a key feature in evolvable architectures. The idea of wrapping is to replace an old component named, for instance, a with a new component b that exploits a to perform its behavior. One can also imagine that b changes/extends the behavior of a .

A simple definition of wrapping can be given inspired by the **Rename** macro defined above. The main difference is that now the information about the behavior of component b is needed as a parameter. We call this definition **NaifWrap**. We will present a refined definition just after. Macro **NaifWrap** $(a, b(S)\{M\}[P], P')$ takes a component a and puts it inside the wrapper b , executing then process P' to make the environment aware of the completion of the operation. As for the renaming operation, we assume that names a and b are different.

$$\begin{aligned} & \mathbf{NaifWrap}(a, b(S)\{M\}[P], P') \\ & \triangleq \nu ack \mathbf{Hide}(a, b, (b(S \cup \{a\})\{M\}[P] \mid a \mathbf{in} b.ack\langle 0 \rangle.0), ack, P') \end{aligned}$$

One may also want to wrap a component a using a wrapper with the same name. In this way in fact an external component will not notice that wrapping has been performed, and can interact with the wrapper as if it was the old component. This is particularly useful when one uses wrapping to extend the interface or the behavior of an existing component.

The naif definition of wrapping above does not work when the wrapper and the wrapped component have the same name. In this case the code of the wrapper must be able to call the old component a , but also to perform self-calls. On the contrary, neither the environment nor the old component a should notice that the wrapping has been performed. In particular, calls to a in the environment should activate the wrapper.

For these reasons we propose below a refined definition of wrapping. The new definition uses both renaming and naif wrap. Macro **Wrap** $(b, a(S)\{M\}[P], P')$ takes component a and wraps it into a wrapper with the same name, which is supposed to use b (different from a) to reference the old component.

$$\begin{aligned} & \mathbf{Wrap}(b, a(S)\{M\}[P], P') \\ & \triangleq \nu ch, ack (\mathbf{Hide}(a, a, \left(\mathbf{Rename}(a, b, ch\langle 0 \rangle.0) \mid \right. \\ & \quad \left. ch(x).\mathbf{NaifWrap}(b, a(S)\{M\}[P], ack\langle 0 \rangle.0) \right), ack, P')) \end{aligned}$$

When the macro is put in parallel with a component a , a is renamed into b and then wrapped. In the definition, channel ch is used to ensure that renaming of a has been completed before performing the actual wrapping.

Update. Updating a component means adding to it some new features. This can be done by adding new methods, new processes or new sub-components.

In COMP, the interface of components is fixed. However adding new methods can be simulated. In fact, it is a particular case of the wrap operation defined above.

Adding new processes and sub-components can also be done, using higher-order communication. This can be programmed in such a way that the updated component can also perform some check to ensure the validity of the update before installing it. In particular, the component to be updated should provide a dedicated method for update. The macro **Update** (a, P) requiring component a to install the update P is defined as:

$$\mathbf{Update}(a, P) \triangleq a.\mathbf{upd}\langle P \rangle$$

and assumes that component a has the form:

$$a(S)\{\text{upd}(X).X \mid M\}[P']$$

The update mechanism could be used to apply one of the previously discussed reconfiguration patterns to subcomponents. In fact, the above patterns can be applied only to parallel components. In case one wants to apply them also to nested components, the update mechanism could be used to forward downward the reconfiguration request. For instance, assume that one process in parallel with a component a wants to rename a component b contained in a to name c . This could be accomplished by using the update mechanism to send to the component a the rename macro:

$$\mathbf{Update}(a, \mathbf{Rename}(b, c, 0)) \mid a(S)\{\text{upd}(X).X \mid M\}[b(S')\{M'\}[P'] \mid P]$$

A more refined macro $\mathbf{Update}(a, c, P)$ may include also a channel name c where to ask for the credentials of the update. In such a way component a may insert some code for checking the validity of the update in between the receipt of the update request and the execution of X .

Links. In many component models in the literature [24,23,31], components come equipped with input ports, output ports, and links connecting input ports to output ports. In our model methods can be considered as interfaces of an input port. However, to keep the calculus focused and manageable, we choose not to include explicit output ports and links as language constructs. Here, we show that output ports and links can be defined by the calculus, thus enabling the programmer to exploit them. In particular, different disciplines of linking ports can be defined and analyzed using the calculus.

The output ports of a component a represent the dependencies that a needs in order to perform its task. Output ports can be seen in our model as virtual methods. Connecting one such method \mathfrak{m}_v to a real method \mathfrak{m}_r (typically, from another component) using a link means that the dependency \mathfrak{m}_v is satisfied by invoking method \mathfrak{m}_r . In other words, a link acts as a forwarder. We encode a link between the output port $a.\mathfrak{m}_v$ and the input port $a'.\mathfrak{m}_r$ as a component (named here b) of the form:

$$b(\emptyset)\{0\}[a(\emptyset)\{\mathfrak{m}_v(x).a'.\mathfrak{m}_r(x)\}\{0\}]$$

Each link is a component with its own name, so that it can be referred to, e.g. remove it when it is no more needed.

We can finally present the macros for creating and deleting links. Macro $\mathbf{Connect}(a.\mathfrak{m}_v, a'.\mathfrak{m}_r, b, P)$ creates a link named b connecting port $a.\mathfrak{m}_v$ to method $a'.\mathfrak{m}_r$ and executing P upon completion. Macro $\mathbf{Disconnect}(b, P)$ removes link b and executes P upon completion.

$$\mathbf{Connect}(a.\mathfrak{m}_v, a'.\mathfrak{m}_r, b, P) \triangleq b(\emptyset)\{0\}[a(\emptyset)\{\mathfrak{m}_v(J).a'.\mathfrak{m}_r(J)\}\{0\}] \mid P$$

$$\mathbf{Disconnect}(b, P) \triangleq \mathbf{Remove}(b, P)$$

The links defined here are quite primitive, in the sense that they are subject to interference. However interferences could be avoided. First one would need to use fresh names to link components. Also, links are attached here to component names. On one hand this ensures that links are not spoiled by wrapping. On the other hand, they are spoiled when the target component is renamed. If this is not the desired behavior, one has to update the links when a component is renamed. This may require keeping track of all the links referring to a given component. We do not consider this issue in more detail here.

Distribution. In COMP, there are no dedicated constructs for modeling physical distribution. However, as we already said, components can also represent physical locations. We can consider for instance a top-level component representing the network, and containing a sub-component for each site. The network component can e.g. be used to implement communication protocols. It can even provide facilities for deployment and re-deployment of components.

A sample configuration is defined below. It includes two sites, Site_1 and Site_2 , running processes P_1 and P_2 respectively. The two sites cannot communicate through method calls. In fact, each of them can just see the network. The network however provides an asynchronous communication mechanism, made available as a global component Net relaying messages from one site to another. We assume to this end that Site_1 (resp. Site_2) listens for incoming messages via method in , and can be reached by calling the method $\text{Net_toS}_1(x)$ (resp. $\text{Net_toS}_2(x)$) provided by the Net component.

$$\begin{array}{l} \text{Network}(\emptyset)\{0\}[\\ \quad \text{Site}_1(\overline{\{\text{Net}\}})\{\text{in}(x).P'_1\}[P_1] \\ \quad | \text{Site}_2(\overline{\{\text{Net}\}})\{\text{in}(x).P'_2\}[P_2] \\ \quad | \text{Net}(\emptyset)\{\text{toS}_1(x).\text{Site}_1\text{-in}(x) \mid \text{toS}_2(x).\text{Site}_2\text{-in}(x)\}\{0\} \\] \end{array}$$

Note that in this example we use the set of names $\overline{\{\text{Net}\}}$ to specify that both sites Site_1 and Site_2 can only communicate with the component Net which encodes the network protocols. Also, the network protocols defined here are trivial (pure forwarding), but they can easily be extended to account correctness checking, buffering or other features.

Re-deployment. Since distribution is defined in terms of our component notion, and the component hierarchy can be dynamically updated at any moment, we can exploit our primitives for dynamic re-deployment.

Suppose we have a running system as above, and that we want to move a component from one location to another, e.g. for efficiency reasons. This can be obtained by adding to the component Network above some code for performing re-deployment. A basic example is given below, where we add a method move to the Network :

$$\text{Network}(\emptyset)\{\text{move}(a, l_1, l_2).a \text{ out } l_1.a \text{ in } l_2.0\}[\dots]$$

Method `move` takes three parameters, the name a of the component to be moved, its current location l_1 and its destination l_2 . Simple forms of re-deployment can be obtained by invoking method `Network_move` with suitable parameters. Clearly, more refined implementations of primitives for re-deployment are possible, keeping into account more complex reconfiguration requests or performing suitable checks on the request and on the state of the system before actually executing the request.

5 Related Work

Components have been introduced as a new programming paradigm in the mid-nineties, as a mean to solve several limitations of the object models [31]. One of the main such limitations is the lack of high level operators for adaptation/evolution, as motivated in [24,25]. Nowadays, many component models exist, distinguished by their definition of the component structure and by the operations provided on them. For instance, the OSGi component model [2] developed by IBM defines its components as a set of objects and classes with some extra information, as which services the component provides, and on which services it depends on. This model thus allows one to add components at runtime, with a constraint solver that checks and solves the dependencies of the added component. Let us note that components in OSGi can only be assembled in a flat structure, while in Fractal [4], developed by INRIA and France-Telecom, they can be assembled into a tree structure. But in contrast to OSGi, in Fractal, the programmer must explicitly specify how the dependencies are solved by using *bindings*, similar to the links presented in our examples.

The main focus of these component models, as well as for many others, like COM [8], JAVA BEANS [30], APPIA [21], CLICK [23], COYOTE [3] or DREAM [14], is practical and implementation support for code packaging and wiring of components (for instance, OSGi is the basis of the Eclipse IDE and its plugin mechanism). Some of the component models allow evolution steps for the components, but do not provide formally defined operators to reason about these steps. Indeed, to meet practical requirements, the models are tied to large programming languages and come together with complex APIs. This makes them inappropriate for the formal investigation of dynamic evolution steps in the context of ABS models. Moreover, even if many of the models are based on an object-oriented language, none of them are concerned with the dynamic behavior or describe the interaction between components and objects. For instance, the FRACTAL model states that communication between components can only occur by using the input and output ports. But because of its implementation in Java, it is possible to use the objects and their methods to make components communicate without an explicit use of ports. Our approach aims to solve these two problems: as our model is defined in the process calculus style it has a precise semantics, and we designed our components to be an extension of ABS object groups, with additional isolation and mobility capabilities: hence the interactions between components and objects are naturally described in our model.

A few component models are formally defined, most of them in the process calculus style. There are, for instance, the M-calculus [28] and the kell-calculus [29], both inspired by Fractal. Other calculi, like the different flavors of the Ambient calculus [6,32,16,5], the join-calculus [11] or the seal calculus [7] use named boxes as a means to structure the program into a tree hierarchy. Moreover, some of these calculi use this structure to control communication and to enable adaptation through the modification of the program structure. Finally, in Oz/K [17], the authors have proposed a core programming language with components as a main feature. These component models, being formally defined, are a better fit for ABS than the previous ones, but still have several limitations. First, only Oz/K integrates objects in its model. The other proposals do not provide any description of how components and objects interact (actually, many of them have no concept of object at all) and so cannot be directly integrated with ABS. Moreover, Oz/K has quite a complex communication pattern, and deals with adaptation via the use of *passivation*, which, as suggested by [15], is a too high level operation to hope for any tool to help proving behavioral properties. Our model is by design simple and close to objects so to solve by construction the limitations of existing formal component models described above.

6 Conclusion

In this paper we have presented COMP, a component model designed to be easily integrated into an object-oriented language like ABS, and defined with a formal semantics in order to be a basis for proofs of correctness and behavioral properties. We also put our model at work on a set of simple patterns for dynamic reconfiguration, showing e.g. that component removal, update and wrapping can easily be defined using the primitives that we propose.

The work presented here opens many possibilities for further development. First, we want a full integration between COMP and ABS. We expect no major difficulties, since COMP has been developed with this aim in mind, and components are designed as extended objects, but a few technical steps are required. First, the semantics of ABS is defined in a purely rewriting style, thus we should adapt the COMP semantics. Then one has to add component definitions by instantiation of suitable component classes (extending object classes). Finally, communication patterns based on channels have to be reformulated in terms of asynchronous method calls and futures.

Another important but orthogonal task that we plan to investigate is a type system for COMP. The main point here is the trade-off between the strong guarantees that the type system should provide (e.g., in terms of absence of message-not-understood errors), and the possibility of typing evolution steps able to change the structure of components.

A last point to be investigated is the introduction of error handling and compensations in the model. This would allow one to manage errors (due, e.g., to inconsistent reconfigurations) ensuring that the whole system can reach a consistent state. Such an approach is under analysis in ABS, but we plan to extend

it to deal with components. In particular, the hierarchy of components may be used as hierarchy for error management. Also, compensations may allow to locally manage the effect of a global reconfiguration.

References

1. Ahrendt, W., Dylla, M.: A system for compositional verification of asynchronous objects. *Science of Computer Programming* (2010) (in press)
2. OSGi Alliance. *Osgi Service Platform, Release 3*. IOS Press, Inc., Amsterdam (2003)
3. Bhatti, N.T., Hiltunen, M.A., Schlichting, R.D., Chiu, W.: Coyote: A system for constructing fine-grain configurable communication services. *ACM Trans. Comput. Syst.* 16(4) (1998)
4. Bruneton, E., Coupaye, T., Leclercq, M., Quema, V., Stefani, J.-B.: The Fractal Component Model and its Support in Java. *Software - Practice and Experience* 36(11-12) (2006)
5. Bugliesi, M., Castagna, G., Crafa, S.: Access control for mobile agents: the calculus of boxed ambients. *ACM. Trans. Prog. Languages and Systems* 26(1) (2004)
6. Cardelli, L., Gordon, A.D.: Mobile Ambients. *Theoretical Computer Science* 240(1) (2000)
7. Castagna, G., Vitek, J., Nardelli, F.Z.: The Seal calculus. *Inf. Comput.* 201(1) (2005)
8. Coulson, G., Blair, G., Grace, P., Joolia, A., Lee, K., Ueyama, J.: OpenCOM v2: A Component Model for Building Systems Software. In: *Proceedings of IASTED Software Engineering and Applications, SEA 2004* (2004)
9. de Boer, F.S., Clarke, D., Johnsen, E.B.: A complete guide to the future. In: De Nicola, R. (ed.) *ESOP 2007*. LNCS, vol. 4421, pp. 316–330. Springer, Heidelberg (2007)
10. Full ABS Modeling Framework, Deliverable 1.2 of project FP7-231 620 (HATS) (March 2011), <http://www.hats-project.eu>
11. Fournet, C., Gonthier, G.: The join calculus: A language for distributed mobile programming. In: Barthe, G., Dybjer, P., Pinto, L., Saraiva, J. (eds.) *APPSEM 2000*. LNCS, vol. 2395, pp. 268–332. Springer, Heidelberg (2002)
12. European Project HATS, <http://www.hats-project.eu>
13. Johnsen, E.B., Owe, O.: An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling* 6(1), 35–58 (2007)
14. Leclercq, M., Quema, V., Stefani, J.-B.: DREAM: a Component Framework for the Construction of Resource-Aware, Configurable MOMs. *IEEE Distributed Systems Online* 6(9) (2005)
15. Lenglet, S., Schmitt, A., Stefani, J.-B.: Howe’s Method for Calculi with Passivation. In: Bravetti, M., Zavattaro, G. (eds.) *CONCUR 2009*. LNCS, vol. 5710, pp. 448–462. Springer, Heidelberg (2009), doi:10.1007/978-3-642-04081-8_30
16. Levi, F., Sangiorgi, D.: Mobile safe ambients. *ACM. Trans. Prog. Languages and Systems* 25(1) (2003)
17. Lienhardt, M., Schmitt, A., Stefani, J.-B.: Oz/k: A kernel language for component-based open programming. In: *GPCE 2007: Proceedings of the 6th International Conference on Generative Programming and Component Engineering*, pp. 43–52. ACM, New York (2007)

18. Liu, X., Kreitz, C., van Renesse, R., Hickey, J., Hayden, M., Birman, K., Constable, R.: Building Reliable, High-Performance Communication Systems from Components. In: Proceedings of the 1999 ACM Symposium on Operating Systems Principles, Kiawah Island, SC (December 1999)
19. Merro, M., Nardelli, F.Z.: Behavioral theory for mobile ambients. *J. ACM* 52(6), 961–1023 (2005)
20. Milner, R., Parrow, J., Walker, J.: A calculus of mobile processes, I and II. *Inform. and Comput.* 100(1), 1–40, 41–77 (1992)
21. Miranda, H., Pinto, A.S., Rodrigues, L.: Appia: A flexible protocol kernel supporting multiple coordinated channels. In: 21st International Conference on Distributed Computing Systems (ICDCS 2001). IEEE Computer Society, Los Alamitos (2001)
22. Montesi, F., Sangiorgi, D.: A model of evolvable components. In: Wirsing, M., Hofmann, M., Rauschmayer, A. (eds.) TGC 2010, LNCS, vol. 6084, pp. 153–171. Springer, Heidelberg (2010), doi:10.1007/978-3-642-15640-3_11
23. Morris, R., Kohler, E., Jannotti, J., Frans Kaashoek, M.: The Click Modular Router. In: ACM Symposium on Operating Systems Principles (1999)
24. Oreizy, P., Medvidovic, N., Taylor, R.N.: Architecture-based runtime software evolution. In: Proceedings of the 20th International Conference on Software Engineering, ICSE 1998, pp. 177–186. IEEE Computer Society, Washington, DC, USA (1998)
25. Oreizy, P., Medvidovic, N., Taylor, R.N.: Runtime software adaptation: framework, approaches, and styles. In: Companion of the 30th International Conference on Software Engineering, ICSE Companion 2008, pp. 899–910. ACM, New York (2008)
26. Sangiorgi, D.: From pi-calculus to higher-order pi-calculus - and back. In: Gaudel, M.-C., Jouannaud, J.-P. (eds.) CAAP 1993, FASE 1993, and TAPSOFT 1993. LNCS, vol. 668, pp. 151–166. Springer, Heidelberg (1993)
27. Schäfer, J., Poetzsch-Heffter, A.: Jacobox: Generalizing active objects to concurrent components. In: D’Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 275–299. Springer, Heidelberg (2010)
28. Schmitt, A., Stefani, J.-B.: The M-calculus: A Higher-Order Distributed Process Calculus. In: Proceedings 30th Annual ACM Symposium on Principles of Programming Languages, POPL (2003)
29. Schmitt, A., Stefani, J.-B.: The Kell Calculus: A Family of Higher-Order Distributed Process Calculi. In: Priami, C., Quaglia, P. (eds.) GC 2004. LNCS, vol. 3267, pp. 146–178. Springer, Heidelberg (2005)
30. Sun Microsystems. JSR 220: Enterprise JavaBeans, Version 3.0 – EJB Core Contracts and Requirements (2006)
31. Szyperski, C.: Component Software, 2nd edn. Addison-Wesley, Reading (2002)
32. Teller, D., Zimmer, P., Hirschhoff, D.: Using Ambients to Control Resources. In: Brim, L., Jančar, P., Křetínský, M., Kučera, A. (eds.) CONCUR 2002. LNCS, vol. 2421, pp. 288–303. Springer, Heidelberg (2002)

Compositional Algorithmic Verification of Software Product Lines^{*}

Ina Schaefer¹, Dilian Gurov², and Siavash Soleimanifard²

¹ Technische Universität Braunschweig, Germany
`i.schaefer@tu-braunschweig.de`

² Royal Institute of Technology, Stockholm, Sweden
`{dilian,siavashs}@csc.kth.se`

Abstract. Software product line engineering allows large software systems to be developed and adapted for varying customer needs. The products of a software product line can be described by means of a *hierarchical variability model* specifying the commonalities and variabilities between the artifacts of the individual products. The number of products generated by a hierarchical model is exponential in its size, which poses a serious challenge to software product line analysis and verification. For an analysis technique to scale, the effort has to be linear in the size of the model rather than linear in the number of products it generates. Hence, efficient product line verification is only possible if *compositional* verification techniques are applied that allow the analysis of products to be *relativized* on the properties of their variation points. In this paper, we propose simple hierarchical variability models (SHVM) with explicit variation points as a novel way to describe a set of products consisting of sets of methods. SHVMs provide a trade-off between expressiveness and a clean and simple model suitable for compositional verification. We generalize a previously developed compositional technique and tool set for the automatic verification of control-flow based temporal safety properties to product lines defined by SHVMs, and prove soundness of the generalization. The desired property relativization is achieved by introducing variation point specifications. We evaluate the proposed technique on a number of test cases.

1 Introduction

System diversity is prevalent in modern software systems. Systems simultaneously exist in many different variants in order to adapt to their application context. Software product line engineering [23] aims at developing a set of systems variants with well-defined commonalities and variabilities by managed reuse in order to decrease time to market and to improve quality. During family engineering reusable core artifacts are developed, that are used to realize the actual products during application engineering.

^{*} This work has been partially supported by the Deutsche Forschungsgemeinschaft (DFG) and the EU project FP7-231620 HATS.

The variability of the artifacts used for building a software product line can be described by a hierarchical variability model. In this model, on each level of hierarchy the commonalities of the product artifacts are specified in a common core, while the variabilities are represented by explicit variation points. Each variation point is associated with a set of variants that represent choices for realizing the variation points in different products. A variant can itself contain commonalities defined in a common core and variabilities specified by variation points introducing a new level of hierarchy.

Product line verification typically aims at establishing that *all* products of a product line satisfy a desired set of properties. The number of products defined by a hierarchical variability model is exponential in the size of the model. This explosion poses serious problems to ensuring the critical product requirements by static analysis or other formal verification techniques, and can render infeasible the verification of product lines by verifying all products individually. Formal verification techniques will only scale if their complexity is linear in the size of the hierarchical variability model rather than linear in the number of products. In order to achieve this scalability, these techniques have to be compositional, allowing to relativize the product properties towards properties of variation points.

In this paper, we generalize a previously developed compositional verification technique (and the corresponding tool set) for the automatic verification of control-flow based temporal safety properties [13,15] to the compositional verification of hierarchically defined product lines. We propose simple hierarchical variability models (SHVM) as a novel way to specify the variability of product artifacts. SHVMs provide a clean and simple model facilitating compositional verification, while they are still sufficiently expressive for capturing variability. In this work, product artifacts consist of sets of public and private methods. In an SHVM, the artifact variability is defined by common core methods and explicit variation points on different hierarchical levels. The properties that can be handled fully automatically specify illegal sequences of method invocations, such as improper usage of API methods, in terms of temporal logic formulas, abstracting from the computed data. Compositionality, and the ability to relativize global SHVM properties on local assumptions for the core methods and the variation points, is achieved by means of maximal flow graphs that are derived algorithmically from the local assumptions. The flow graphs replace the assumptions when verifying global properties. The local specifications of core methods are verified by extracting flow graphs from the method implementations and model checking the induced behaviors against their specification.

The presented approach is one of the first compositional verification techniques for software product lines. It allows to guarantee efficiently that all products of a product line satisfy certain desired control-flow based safety properties. With respect to model checking behavioral properties of product lines, only Blundell et al. [4] and Liu et al. [20] propose compositional verification techniques based on assume-guarantee style reasoning for product features. Other model checking approaches for product lines [8,10,18,5] use a monolithic model of the

complete product line such that they face severe state–space explosion problems since all possible products are analyzed in the same analysis step.

The paper is organized as follows. In Section 2, we present SHVMs to hierarchically represent product lines. In Section 3, we describe the foundations of our compositional verification technique. In Section 4, we present the compositional verification procedure for product lines and prove its soundness. In Section 5, we present tool support and an evaluation of the compositional verification technique. In Section 6, we review related work and conclude the paper in Section 7.

2 Hierarchical Variability Modelling

A *product* in the context of this work is defined by a set of methods. Products are not necessarily closed, i.e., they may still require additional methods such as API methods. A method m from a set of methods $Meth$ is understood as a method definition, consisting of a method name, the types of the return value and the parameters, and its implementation (method body). The methods of a product are partitioned into *public* and *private* methods. Public methods are visible to the outside of the product, while private methods are only visible within products and can be viewed as a means of implementing the public methods. For a product, the methods defined in the product are called *provided*, while the called methods that are not provided themselves are referred to as *required*.

A *product line* PL is defined as a set of method sets $PL \subseteq 2^{Meth}$ and can be represented by a *hierarchical variability model*, with the common methods of all products captured by a *core* set of methods separated into public and private methods. The differences between the products are represented by *variation points*. To each variation point, a set of *variants* is attached. The variants represent different possibilities to realize the variability described by this variation point. A variant can either comprise a set of core methods or be a hierarchical variability model itself, i.e., consisting of core set of methods and a set of variation points. A product is derived by resolving the variabilities, i.e., by selecting variants at the variation points on all levels of hierarchy. An example is given later in this section in Figure 1.

Hierarchical variability modeling captures the variability of the artifacts that are used to build the products, called solution space variability in 7. In this work, hierarchical variability modeling describes the variability of the methods implementing single products. This is in contrast to problem space variability 7 which is mainly represented in terms of product features. Product features denote a user-visible product functionality and are merely labels without inherent semantical meaning. The valid combinations of product features can be described by feature models 16 and correspond to the valid member products. The tree-hierarchy in feature models usually describes the sub/super-feature relationship between product features, while the hierarchy in hierarchical variability models refers to the commonality and variability of the solution space artifacts.

In this paper, we introduce a variant of the hierarchical variability modelling approach called *simple* hierarchical variability model (SHVM). An SHVM is a hierarchical variability model that requires exactly one variant to be selected at each variation point to obtain a product. In an SHVM, there is no means for defining constraints between different variants and variation points to represent that the selection of a variant at one variation point requires a specific variant at another variation point to be selected, thus restricting the number of derivable products. These simplifications constitute a trade-off between providing an expressive representation of product variability and a clean model that allows straight-forward application of the compositional verification procedure described in Section 4. This trade-off is discussed at the end of this section.

Definition 1 (Simple Hierarchical Variability Model). A simple hierarchical variability model (SHVM) \mathcal{S} is inductively defined as:

- (i) a ground model consisting of a core set of methods $M_C = (M_{pub}, M_{priv})$, partitioned into public and private methods $M_{pub}, M_{priv} \subseteq Meth$, or
- (ii) a pair $(M_C, \{VP_1, \dots, VP_N\})$, where M_C is defined as above and where $\{VP_1, \dots, VP_N\}$ is a non-empty set of variation points. A variation point $VP_i = \{\mathcal{S}_{i,j} \mid 1 \leq j \leq k_i\}$ is a non-empty set of SHVMs. The members of a variation point are called variants.

The *variant interface* of a pair $(M_C, \{VP_1, \dots, VP_N\})$ is defined as a pair of public required and public provided methods. The set of public provided methods is the union of all sets of public provided methods in the core methods and the variation points. The set of public required methods is the union of all sets of public methods required by the core methods and by the variation points without the methods provided by the core methods or another variation point.

We assume the following two *well-formedness* constraints on SHVMs. First, all variants attached to a variation point have to provide and require the same sets of public methods. This pair of public required and provided methods is called the *variation point interface*. The constraint guarantees that all variants offer the same functionality in terms of the provided public methods while the implementation of the public methods may differ in the variant's private methods. Second, in order to enforce that a derivable product does not contain several methods with the same name, it is required that the provided methods in each variation point interface are disjoint from each other and the core method set.

Example 1. As a running example throughout this paper, we consider a product line of cash desks that is a simplified version of the trading system product line case study proposed in [24]. The cash desks process purchases by retrieving the prices for all items to be purchased and calculating the total price. After the customer has paid, a receipt is printed and the stock is updated accordingly. The commonality of all cash desks is that every purchase is processed following the same process. However, the cash desks differ in the way how the items are entered. Some cash desks allow entering products using a keyboard, others only provide a scanner, and a third group provides both options which can be chosen

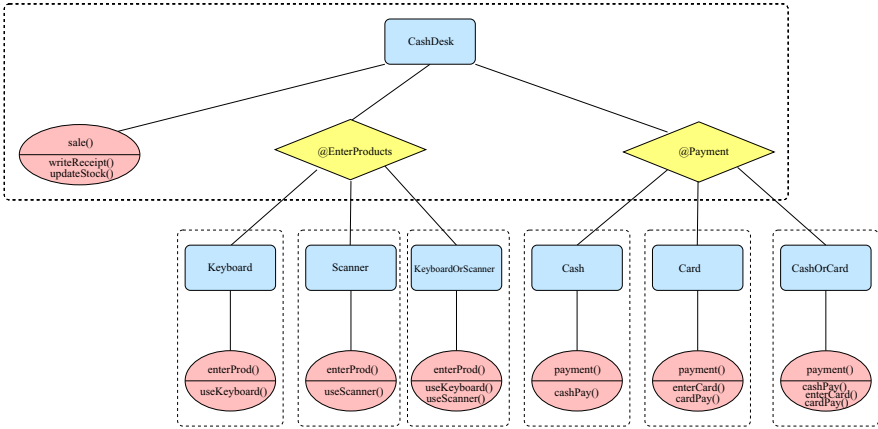


Fig. 1. The Cashdesk SHVM

by the cashier. Payment at some cash desks can only be made in cash. Other cash desks only accept credit cards, while a third group allows the choice between cash and credit card payment. This set of cash desks is defined by an SHVM as follows:

$$\text{CashDesk} = ((\{\text{sale}\}, \{\text{updateStock}, \text{writeReceipt}\}), \{\text{@EnterProducts}, \text{@Payment}\})$$

$$\text{where } \text{@EnterProducts} = \{\text{Keyboard}, \text{Scanner}, \text{KeyboardOrScanner}\}$$

$$\text{@Payment} = \{\text{Cash}, \text{Card}, \text{CashOrCard}\}$$

$$\text{and Keyboard} = (\{\text{enterProd}\}, \{\text{useKeyboard}\})$$

$$\text{Scanner} = (\{\text{enterProd}\}, \{\text{useScanner}\})$$

$$\text{KeyboardOrScanner} = (\{\text{enterProd}\}, \{\text{useScanner}, \text{useKeyboard}\})$$

$$\text{Cash} = (\{\text{payment}\}, \{\text{cashPay}\})$$

$$\text{Card} = (\{\text{payment}\}, \{\text{enterCard}, \text{cardPay}\})$$

$$\text{CashOrCard} = (\{\text{payment}\}, \{\text{cashPay}, \text{enterCard}, \text{cardPay}\})$$

The common purchase process of all cash desks is modeled by the public core method `sale`. The private methods `updateStock` and `writeReceipt` represent internal details of the sale process. The two variation points `@EnterProducts` and `@Payment` represent the variabilities of the cash desks. The variation point `@EnterProducts` has the associated variants `Keyboard`, `Scanner` and `KeyboardOrScanner` for entering product by keyboard, by scanner or providing both options. Both provide the public method `enterProd` that is internally realized by the different private methods `useKeyboard`, `useScanner` or their combination. Similarly, the variation point `@Payment` has the associated variants `Cash`, `Card` and `CashOrCard` that provide the public method `payment` which is internally realized by different private methods in the respective variants.

An SVHM can be seen as a tri-partite directed *graph* having an SHVM-node as root, where SHVM-nodes have one core methods leaf child (split in public

and private methods) and optional VP–node children that have two or more SHVM–node children. For the cashdesk example, a graphical presentation is shown in Figure 1. In the figure, SHVM–nodes are depicted by rounded boxes, core methods nodes by ovals, and VP–nodes by diamonds. The dotted rounded boxes depict what we call *modules* of the SHVM, defining the boundaries between SHVMs at different levels of hierarchy. The *size* of an SHVM is defined as the number of modules in its graph.

An SHVM *induces* a set of products P through all possible ways of resolving the variabilities of the SHVM. Variability resolution means to recursively select exactly one variant for each variation point. The set of products induced by a ground model containing only core methods is the singleton set comprising the set of core methods (and, thus, representing one product). The set of products induced by a variation point is the union of the product sets induced by its variants. Finally, the set of products induced by an SHVM with a non–empty set of variation points is the set of all products consisting of the core methods and of exactly one product from the set induced by each variation point.

Definition 2 (Variability Resolution). *Let \mathcal{S} be an SHVM as defined above. The set $\text{products}(\mathcal{S}) \subseteq 2^{\text{Meth}}$ induced by \mathcal{S} is inductively defined as follows:*

$$\begin{aligned} \text{products}(M_C) &= \{M_C\} \\ \text{products}(VP) &= \bigcup_{S \in VP} \text{products}(S) \\ \text{products}(M_C, \{VP_1, \dots, VP_N\}) &= \left\{ M_C \cup \bigcup_{1 \leq i \leq N} M_i \mid M_i \in \text{products}(VP_i) \right\} \end{aligned}$$

Example 2. The SHVM defined in Example 1 induces the products:

$$\text{products}(\text{CashDesk}) = \{P_1, P_2, P_3, P_4, P_5, P_6, P_7, P_8, P_9\}$$

where:

$$\begin{aligned} P_1 &= \left\{ \text{sale, updateStock, writeReceipt, enterProd}_{\text{Keyboard}}, \right. \\ &\quad \left. \text{useKeyboard, payment}_{\text{Cash}}, \text{cashPay} \right\} \\ P_2 &= \left\{ \text{sale, updateStock, writeReceipt, enterProd}_{\text{Scanner}}, \right. \\ &\quad \left. \text{useScanner, payment}_{\text{Cash}}, \text{cashPay} \right\} \\ P_3 &= \left\{ \text{sale, updateStock, writeReceipt, enterProd}_{\text{KeyboardOrScanner}}, \right. \\ &\quad \left. \text{useKeyboard, useScanner, payment}_{\text{Cash}}, \text{cashPay} \right\} \\ P_4 &= \left\{ \text{sale, updateStock, writeReceipt, enterProd}_{\text{Keyboard}}, \right. \\ &\quad \left. \text{useKeyboard, payment}_{\text{Card}}, \text{enterCard, cardPay} \right\} \\ P_5 &= \left\{ \text{sale, updateStock, writeReceipt, enterProd}_{\text{Scanner}}, \right. \\ &\quad \left. \text{useScanner, payment}_{\text{Card}}, \text{enterCard, cardPay} \right\} \\ P_6 &= \left\{ \text{sale, updateStock, writeReceipt, enterProd}_{\text{KeyboardOrScanner}}, \right. \\ &\quad \left. \text{useKeyboard, useScanner, payment}_{\text{Card}}, \text{enterCard, cardPay} \right\} \\ P_7 &= \left\{ \text{sale, updateStock, writeReceipt, enterProd}_{\text{Keyboard}}, \right. \\ &\quad \left. \text{useKeyboard, payment}_{\text{CashOrCard}}, \text{cashPay, enterCard, cardPay} \right\} \end{aligned}$$

$$P_8 = \left\{ \begin{array}{l} \text{sale, updateStock, writeReceipt, enterProd}_{\text{Scanner}}, \\ \text{useScanner, payment}_{\text{CashOrCard}}, \text{cashPay, enterCard, cardPay} \end{array} \right\}$$

$$P_9 = \left\{ \begin{array}{l} \text{sale, updateStock, writeReceipt, enterProd}_{\text{KeyboardOrScanner}}, \\ \text{useKeyboard, useScanner, payment}_{\text{CashOrCard}}, \\ \text{cashPay, enterCard, cardPay} \end{array} \right\}$$

To disambiguate methods with the same name, but coming from different variants, we add as subscript the name of the parent SHVM-node, for instance, $\text{enterProd}_{\text{Keyboard}}$ refers to the method enterProd of the variant Keyboard .

For a given SHVM, let AND and OR denote the maximal branching factors at SHVM and variation point nodes, respectively, and let ND be its nesting depth. The number of products induced by the SHVM is bound by $OR \frac{AND \cdot (AND^{ND} - 1)}{AND - 1}$ and is thus exponential in the size of the SHVM, which is bound by $\frac{(OR \cdot AND)^{(ND+1)} - 1}{OR \cdot AND - 1}$. These bounds are obtained in a routine fashion by solving the corresponding recurrence relations. Notice that in SHVMs with a small nesting depth as in the example above, the exponential blow-up in the number of products is not observed: With branching factors of 3 and a nesting depth of 1, we have at most 9 products, but 7 modules. However, adding just another level of hierarchy, e.g., variability in the accepted type of cards, immediately results in an explosion (see Section 5).

SHVMs are a simplification of hierarchical variability modeling supporting a straight-forward application of compositional reasoning with the following consequences to the expressiveness for product variability. In SHVMs, exactly one variant has to be selected at every variation point. If a combination of variants (including optional variants) should be selectable, the combination has to be modeled as an additional variant associated to this variation point. In most cases (and also in the example in this section), combinations of variants require additional glue code for the cooperation of the different behaviors such that combinations have to be represented as separate variants anyway. SHVMs do not allow requires/excludes constraints between variants. These constraints restrict the set of possible products that can be derived from a hierarchical variability model. The removal of these constraints results in an SHVM which defines products that would not exist otherwise. The requirement that all variants associated to a variation point have the same interface restricts the method variability of the variants. This can be alleviated to some extent by adding required methods to the interface, although they are not called by the variant, and adding dummy implementations for provided methods.

3 A Framework for Compositional Verification

This section outlines the theoretical framework for verification of temporal safety properties upon which our compositional verification technique for product lines (described in the next section) is based. It relies on our earlier work on compositional verification (see e.g. [13, 12]).

Program Model. In order to reason algorithmically about sequences of method invocations, we abstract the set of methods defining our program by ignoring all data. An initialized model serves as an abstract representation of a program's structure and behavior.

Definition 3 (Model). *A model is a (Kripke) structure $\mathcal{M} = (S, L, \rightarrow, A, \lambda)$ where S is a set of states, L a set of labels, $\rightarrow \subseteq S \times L \times S$ a labeled transition relation, A a set of atomic propositions, and $\lambda : S \rightarrow \mathcal{P}(A)$ a valuation, assigning to each state s the set of atomic propositions that hold in s . An initialized model is a pair (\mathcal{M}, E) with \mathcal{M} a model and $E \subseteq S$ a set of initial states.*

A *method graph* is an instance of an initialized model which is obtained by ignoring all data from a method implementation. A *flow graph* is a collection of *method graphs*, one for each method of the program. It is a standard model for the analysis of control flow based properties, see e.g. [3].

Definition 4 (Method graph). *Let Meth be a countably infinite set of methods names. A method graph for method $m \in \text{Meth}$ over a set of method names $M \subseteq \text{Meth}$ is an initialized model (\mathcal{M}_m, E_m) where $\mathcal{M}_m = (V_m, L_m, \rightarrow_m, A_m, \lambda_m)$ is a finite model and $E_m \subseteq V_m$ is a non-empty set of entry points of m . V_m is the set of control nodes of m , $L_m = M \cup \{\varepsilon\}$, $A_m = \{m, r\}$, and $\lambda_m : V_m \rightarrow \mathcal{P}(A_m)$ so that $m \in \lambda_m(v)$ for all $v \in V_m$ (i.e., each node is tagged with its method name). The nodes $v \in V_m$ with $r \in \lambda_m(v)$ are return points.*

Note that methods according to the above definition can have multiple entry points. Flow graphs that are extracted from a program source have single entry points, but the maximal models that we generate for compositional verification can have multiple entry points.

Every flow graph \mathcal{G} is equipped with an *interface* $I = (I^+, I^-)$, denoted $\mathcal{G} : I$, where $I^+, I^- \subseteq \text{Meth}$ are the *provided* and *externally required* methods, respectively. Interfaces are needed when constructing maximal flow graphs. A flow graph is *closed* if its interface does not require any methods, and it is *open* otherwise. Flow graph *composition* is defined as the disjoint union \uplus of their method graphs.

Example 3. Figure 2 shows a simple Java class and the (simplified) flow graph it induces. It consists of two method graphs, for method `even` and method `odd`, respectively. Entry nodes are depicted as usual by incoming edges without source. Its interface is $(\{\text{even}, \text{odd}\}, \emptyset)$, thus the flow graph is closed.

Flow graph *behavior* is also defined as an instance of an initialized model, induced through the flow graph structure. We use transition label τ for internal transfer of control, m_1 `call` m_2 for the invocation of method m_2 by method m_1 when method m_2 is provided by the program and m_1 `call!` m_2 when method m_2 is external, and m_2 `ret` m_1 respectively m_2 `ret?` m_1 for the corresponding return from the call.

Definition 5 (Behavior). *Let $\mathcal{G} = (\mathcal{M}, E) : (I^+, I^-)$ be a flow graph such that $\mathcal{M} = (V, L, \rightarrow, A, \lambda)$. The behavior of \mathcal{G} is defined as an initialized model $b(\mathcal{G}) =$*

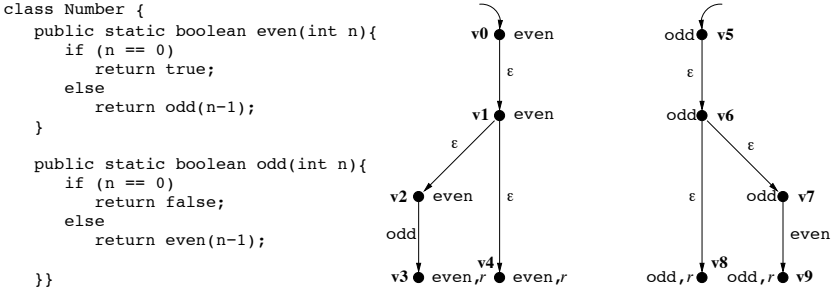


Fig. 2. A simple Java class and its flow graph

(\mathcal{M}_b, E_b) , where $\mathcal{M}_b = (S_b, L_b, \rightarrow_b, A_b, \lambda_b)$, such that $S_b = (V \cup I^-) \times V^*$, i.e., states are pairs of control points v or required method names m , and stacks σ , $L_b = \{m_1 k m_2 \mid k \in \{\text{call}, \text{ret}\}, m_1, m_2 \in I^+\} \cup \{m_1 \text{call! } m_2 \mid m_1 \in I^+, m_2 \in I^-\} \cup \{m_2 \text{ret? } m_1 \mid m_1 \in I^+, m_2 \in I^-\} \cup \{\tau\}$, $A_b = A$, $\lambda_b((v, \sigma)) = \lambda(v)$ and $\lambda_b((m, \sigma)) = m$, and $\rightarrow_b \subseteq S_b \times L_b \times S_b$ is defined by the following rules:

$$\begin{aligned}
[\text{transfer}] \quad & (v, \sigma) \xrightarrow{\tau} (v', \sigma) && \text{if } m \in I^+, v \xrightarrow{\varepsilon}_m v', v \models \neg r \\
[\text{call}] \quad & (v_1, \sigma) \xrightarrow{m_1 \text{ call } m_2} (v_2, v_1 \cdot \sigma) && \text{if } m_1, m_2 \in I^+, v_1 \xrightarrow{m_2}_{m_1} v'_1, v_1 \models \neg r, \\
& && v_2 \models m_2, v_2 \in E \\
[\text{ret}] \quad & (v_2, v_1 \cdot \sigma) \xrightarrow{m_2 \text{ ret } m_1} (v_1, \sigma) && \text{if } m_1, m_2 \in I^+, v_2 \models m_2 \wedge r, v_1 \models m_1 \\
[\text{call!}] \quad & (v_1, \sigma) \xrightarrow{m_1 \text{ call! } m_2} (m_2, v_1 \cdot \sigma) && \text{if } m_1 \in I^+, m_2 \in I^-, v_1 \xrightarrow{m_2}_{m_1} v'_1, v_1 \models \neg r \\
[\text{ret?}] \quad & (m_2, v_1 \cdot \sigma) \xrightarrow{m_2 \text{ ret? } m_1} (v_1, \sigma) && \text{if } m_1 \in I^+, m_2 \in I^-, v_1 \models m_1
\end{aligned}$$

The set of initial states is defined by $E_b = E \times \{\varepsilon\}$, where ε denotes the empty sequence over $V \cup I^-$.

Notice that return transitions always hand back control to the caller of the method. Calls to external methods are modeled with an intermediate state, from which only an immediate return is possible. In this way possible callbacks from external methods are not captured in the behavior. This simplification is justified, since we abstract away from data in the model and the behavior is thus context-free, but has to be kept in mind when writing specifications; in particular one cannot specify that callbacks are not allowed.

Example 4. Consider the flow graph of Example 3. One example run through its (branching, infinite-state) behavior, from an initial to a final configuration, is:

$$\begin{aligned}
(v_0, \varepsilon) \xrightarrow{\tau} (v_1, \varepsilon) \xrightarrow{\tau} (v_2, \varepsilon) \xrightarrow{\text{even call odd}} (v_5, v_3) \xrightarrow{\tau} (v_6, v_3) \xrightarrow{\tau} \\
(v_8, v_3) \xrightarrow{\text{odd ret even}} (v_3, \varepsilon)
\end{aligned}$$

Now, consider just the method graph of method `even` as an open flow graph, having interface $(\{\text{even}\}, \{\text{odd}\})$. The *local contribution* of method `even` to the above global behavior is the following run:

$$(v_0, \varepsilon) \xrightarrow{\tau} (v_1, \varepsilon) \xrightarrow{\tau} (v_2, \varepsilon) \xrightarrow{\text{even call! odd}} (\text{odd}, v_3) \xrightarrow{\text{odd ret? even}} (v_3, \varepsilon)$$

An alternative way to express flow graph behavior is by means of *pushdown systems* (PDS). We exploit this by using pushdown system model checking to verify behavioral properties, see [25].

We refine this program model to allow an explicit partitioning of method names into *public* and *private* ones, and introduce the notions of public interface and public behavior in order to abstract away from private methods which are used as a means of implementing the desired public behavior. On the flow graph level, such an abstraction is accomplished through inlining of private methods. For details the reader is referred to [13].

Specification. The specification language for behavioral properties we use here is the safety–fragment of *Linear Temporal Logic* (LTL) that uses the weak version of until [1].

Definition 6 (Safety LTL). *The formulae of sLTL are inductively defined by:*

$$\phi ::= p \mid \neg p \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \mathbf{X} \phi \mid \mathbf{G} \phi \mid \phi_1 \mathbf{W} \phi_2$$

where $p \in A_b$ denotes the set of atomic propositions.

Satisfaction on states $(\mathcal{M}_b, s) \models \phi$ is defined in the standard fashion (see e.g. [28]) as validity of ϕ over all runs starting from state $s \in S_b$ in model \mathcal{M}_b . For instance, formula $\mathbf{X} \phi$ holds of state s in model \mathcal{M}_b if ϕ holds in the second state of every run starting from s , while $\phi \mathbf{W} \psi$ holds in s if for every run starting in s , either ϕ holds in all states of the run, or ψ holds in some state of the run and ϕ holds in all previous states. Satisfaction of a formula ϕ in flow graph \mathcal{G} with behavior $b(\mathcal{G}) = (\mathcal{M}_b, E_b)$ is defined as satisfaction of ϕ on all initial states $s \in E_b$.

Satisfaction is generalized on product lines in the obvious way: A product line described by a variability model \mathcal{S} satisfies a formula ϕ if the behavior $b(\mathcal{G}_p)$ of the flow graph \mathcal{G}_p of every product $p \in \text{products}(\mathcal{S})$ satisfies ϕ .

Compositional Verification. Our method for *compositional verification* is based on the construction of maximal flow graphs for properties of sets of methods. For a given property ψ and interface I consisting of provided and required methods, consider the class of all flow graphs with interface I satisfying ψ . A *maximal flow graph* for ψ and I is a flow graph $\text{Max}(\psi, I)$ that satisfies exactly those properties that hold for all members of the class. Thus, the maximal flow graph can be used as a representative of the class for the purpose of checking properties. Using maximal models for compositional verification was first proposed in [11] for finite–state systems, and was generalized for flow graphs in [13,12].

¹ The theoretical underpinings of our compositional verification framework are actually based on a slightly more expressive specification language, namely *simulation logic*, the fragment of the modal μ -calculus [17] with boxes and greatest fixed–points only (for details see again [13]).

The main principle of compositional verification based on maximal flow graphs can be presented, for a system that is partitioned into k sets of methods, as a proof rule with $k + 1$ premises:

$$\frac{\mathcal{G}_1 \models \psi_1 \quad \cdots \quad \mathcal{G}_k \models \psi_k \quad \biguplus_{i=1, \dots, k} \text{Max}(\psi_i, I_i) \models \phi}{\biguplus_{i=1, \dots, k} \mathcal{G}_i \models \phi} \quad (1)$$

The principle states that the composition of the sets of methods with the respective interfaces $\mathcal{G}_1 : I_1, \dots, \mathcal{G}_k : I_k$ satisfies a global property ϕ if for some local properties ψ_i satisfied by the corresponding sets of methods \mathcal{G}_i , the composition of the maximal flow graphs for ψ_i and I_i satisfies property ϕ .

As we prove in [13], the rule is sound and complete when interfaces describe all provided and required methods, and is sound in the context of the private method abstraction mentioned earlier.

4 Compositional Verification of SHVMs

In this section we propose a compositional reasoning approach that is linear in the number of modules in the SHVM description of the product line, rather than linear in the number of generated products (which is exponential in the number of modules). This approach is an instantiation of the compositional verification principle presented above to SHVMs.

For every module $(M_C, \{VP_1, \dots, VP_N\})$ in the SHVM, a specification has to be provided in order to allow for compositional verification. This comprises a specification for every public method $m \in M_{pub}$ by a public behavioral property ψ_m and a public interface $I_m = (I_m^+, I_m^-)$ declaring the names of the publicly provided and required methods, a specification for every variation point VP_i by a behavioral property ψ_{VP_i} and a public interface I_{VP_i} , and a specification of the SHVM node itself by a behavioral property ϕ and a public interface I . The SHVM nodes of variants attached to a variation point inherit the corresponding variation point specification. The top-level SHVM is specified by the global product property that is to be verified. Our verification procedure for SHVMs is as follows.

VERIFICATION PROCEDURE. For every module $(M_C, \{VP_1, \dots, VP_N\})$ of the SHVM, perform the following two independent tasks:

- (i) For every public method $m \in M_{pub}$, extract the method graph \mathcal{G}_m from the implementation of m , then inline the already extracted graphs of the private methods, and finally model check the resulting method graph \mathcal{G}'_m against the specification ψ_m of m to establish $\mathcal{G}'_m \models \psi_m$. For the latter, we apply standard finite-state model checking.
- (ii) For all public methods $m \in M_{pub}$ with specification (I_m, ψ_m) , construct the maximal method graphs $\text{Max}(\psi_m, I_m)$, and for all variation points VP_i with

specification (I_{VP_i}, ψ_{VP_i}) , construct the maximal flow graphs $Max(\psi_{VP_i}, I_{VP_i})$. Then, compose the constructed graphs, resulting in flow graph \mathcal{G}_{Max} , and model check the latter against the SHVM property ϕ , i.e.,

$$\left(\bigsqcup_{m \in M_{pub}} Max(\psi_m, I_m) \sqcup \bigsqcup_{1 \leq i \leq N} Max(\psi_{VP_i}, I_{VP_i}) \right) \models \phi \quad (2)$$

For properties given in sLTL, we represent the behavior of \mathcal{G}_{Max} as a PDS and use standard PDS model checking.

The presented verification procedure is *sound*, as established by the following theorem.

Theorem 1. *Let \mathcal{S} be an SHVM with global property ϕ . If the verification procedure succeeds for \mathcal{S} , then $p \models \phi$ for all its products $p \in products(\mathcal{S})$.*

Proof. The proof is by induction on the structure of \mathcal{S} . For the base case, let \mathcal{S} be a ground model, i.e., a core set of methods $M_C = (M_{pub}, M_{priv})$ with no variation points. Assume the verification procedure succeeds for \mathcal{S} . It has then established:

- (i) $\mathcal{G}'_m \models \psi_m$ for all public methods $m \in M_{pub}$, and
- (ii) $\bigsqcup_{m \in M_{pub}} Max(\psi_m, I_m) \models \phi$

From these, and by soundness of rule (II) refined for private method abstraction, it follows $M_C \models \phi$. Since $products(\mathcal{S}) = \{M_C\}$ in this case, we have $p \models \phi$ for all $p \in products(\mathcal{S})$.

For the induction step, let \mathcal{S} be a non-ground model $(M_C, \{VP_1, \dots, VP_N\})$ with variation points $VP_i = \{\mathcal{S}_{i,j} \mid 1 \leq j \leq k_i\}$, where k_i is the number of variants of VP_i . Further, let (ψ_{VP_i}, I_{VP_i}) be the specification of VP_i . Assume the result for all $\mathcal{S}_{i,j}$ (induction hypothesis). Next, assume that the verification procedure succeeds for \mathcal{S} . The following has then been established for the top-level module:

- (i) $\mathcal{G}'_m \models \psi_m$ for all public methods $m \in M_{pub}$, and
- (ii) $\left(\bigsqcup_{m \in M_{pub}} Max(\psi_m, I_m) \sqcup \bigsqcup_{1 \leq i \leq N} Max(\psi_{VP_i}, I_{VP_i}) \right) \models \phi$

By the assumption, the verification procedure has also succeeded for all $\mathcal{S}_{i,j}$. Thus, by the induction hypothesis, and since the SHVM nodes of variants attached to a variation point inherit the corresponding variation point specification, we have:

$$\forall i : 1 \leq i \leq N. \forall j : 1 \leq j \leq k_i. \forall p \in products(\mathcal{S}_{i,j}). p \models \psi_{VP_i}$$

By Definition 2 we have $products(VP_i) = \bigcup_{1 \leq j \leq k_i} products(\mathcal{S}_{i,j})$, and hence:

- (iii) $\forall i : 1 \leq i \leq N. \forall p \in products(VP_i). p \models \psi_{VP_i}$

Also by Definition 2, we know that every product p of \mathcal{S} is the union of a core M_C and exactly one subproduct from every variation point. Due to (i), the public methods of M_C , after inlining the private ones, meet their respective specifications. Similarly, by (iii), all subproducts meet their respective specifications. Finally, by (ii) and from soundness of rule (II) refined for private method abstraction follows that $p \models \phi$. This concludes the proof. \square

The total number of verification tasks needed to establish the global product line property is, thus, equal to the number of modules, since we have to complete one verification task per module. In contrast, the number of products is exponential in the number of modules.

Example 5. To illustrate our compositional verification approach, we use the cashdesk product line described in Example 1. The global behavioral property we want to verify is informally stated as follows:

The entering of products has to be finished before the payment process has started.

Taking into account the distribution of functionality to methods intended by the variability model from the example, the specification can be approximated as:

If control starts in method `sale`, it cannot reach method `payment` before it has already been in method `enterProd` and then back in `sale`.

In terms of the (global) behavior of the flow graphs of the products induced by the product line, this property can be formalized in sLTL as follows:

$$\varphi_{CD} = \text{sale} \rightarrow (\neg \text{payment} \text{ W } (\text{enterProd} \wedge r \wedge \text{X sale}))$$

where the subformula $\text{enterProd} \wedge r \wedge \text{X sale}$ captures a return from `enterProd` to `sale`.

First, we have to specify all public core methods and variation points of the cashdesk SHVM. The specification of the `sale` method and the `@EnterProd` and `@Payment` variation points are as follows:

- The interface of method `sale` is $I_{\text{sale}} = (\{\text{sale}\}, \{\text{enterProd}, \text{payment}\})$. In order to entail the global property, the local behavioral property that method `sale` (or, more precisely, its method graph as an open flow graph) has to satisfy is that it has to have invoked method `enterProd` and returned from the call before it can invoke method `payment`, after the return from which no more methods are invoked. Formally, this can be expressed by the sLTL formula:

$$\varphi_{\text{sale}} = \text{sale} \text{ W}' \text{enterProd} \text{ W}' \text{sale} \text{ W}' \text{payment} \text{ W}' (\text{G sale})$$

where the derived temporal operator $\phi \text{ W}' \psi$ abbreviates $\phi \wedge (\phi \text{ W } \psi)$ and is by convention right-associative.

- The interface of variation point `@EnterProducts` is $I_{EP} = (\{\text{enterProd}\}, \{\text{payment}\})$. The property required for the variation point is that the `enterProd` method never calls the `payment` method, neither directly nor via a call to one of its non-public methods. Formally, this property can be expressed by the formula²:

$$\varphi_{EP} = \mathbf{G} \neg \text{payment}$$

- The interface of variation point `@Payment` is $I_P = (\{\text{payment}\}, \{\text{enterProd}\})$. Similarly to the variation point above, the property required for this variation point is that the `payment` method never calls the `enterProd` method:

$$\varphi_P = \mathbf{G} \neg \text{enterProd}$$

The variants `Keyboard`, `Scanner`, `KeyboardOrScanner`, `Cash`, `Card` and `CashOrCard` inherit the specification of their SHVM node from the respective variation point specification. The specification of the public methods `enterProd` and `payment` is similar to the specification of the `@EnterProd` and `@Payment` variation points.

Next, we have to verify that all public methods satisfy their behavioral property. For the `sale` method, we have to inline the private methods `writeReceipt` and `updateStock` to obtain the method graph of the sale method. Then we check that the method graph satisfies the property φ_{sale} by finite-state model checking. Similarly, we verify the `enterProd` and `payment` methods defined in the variants `Keyboard`, `Scanner`, `KeyboardOrScanner`, `Cash`, `Card` and `CashOrCard`.

Finally, we have to establish that all SHVMs satisfy their SHVM specification. For the top-level SHVM, we construct the maximal models for the specifications of the variation points `@EnterProducts` and `@Payment` and for the public method φ_{sale} , and model check φ_{CD} against the composition of these maximal models. The properties of the variants `Keyboard`, `Scanner`, `KeyboardOrScanner`, `Cash`, `Card` and `CashOrCard` are easy to verify because each of them contains only one public method. A maximal model for the specification of this public method is constructed and checked against the inherited variation point property.

5 Tool Support and Evaluation

PROMOVER [26] is a fully automated tool for the procedure-modular verification of control flow temporal safety properties of Java programs³. It supports compositional verification by relativizing the correctness of a global program property on properties of individual methods and their public interfaces. All interfaces, local and global properties are provided to the tool as assertions in the form of program annotations. PROMOVER accepts a JML-like syntax for

² This and the following property would trivialise if we specified the set of required methods to be empty. For now, however, our tool does not check interfaces.

³ PROMOVER is available via the web interface www.csc.kth.se/~siavashs/ProMoVer

```

/**                                     /**@variant: Keyboard
* @variation_point:                   * @variant_interface:
*   EnterProd                          *   provided enterProd()
* @variation_point_interface:         * @variation_points:
*   provided enterProd                */
* @variation_point_ltl_prop:
*   G ! payment                       /**@core: Keyboard
* @variants:                           * @local_interface:
*   Keyboard, Scanner,                 *   required
*   KeyboardOrScanner                  * @local_ltl_prop: G ! payment
*/                                      */
                                        public int enterProd(){
                                        ...

```

Fig. 3. Annotations for variation point `@EnterProd` and its variant `Keyboard`

annotations (cf. [19]) as special comments called *pragmas*. To simplify the specification of local properties, PROMOVER provides a facility for extracting local properties from source code. Further, it provides a proof storage and reuse mechanism which stores flow graphs, maximal models and model checking results and reuses these the next time the same program is verified. To reuse the stored information, PROMOVER checks for each method of the program: if the source code of the method has not changed, the stored flow graph of the method is used, if a local specification has not changed the stored maximal model for the specification is used. Further, it provides users with a library of global properties which contains platform as well as application specific properties. For details about PROMOVER, the reader is referred to [27].

We have adapted PROMOVER for verifying properties of SHVMs according to the compositionality principle described in Section 4. For this adaptation, we have extended the annotation language to support the definition of core methods, variants and variation points and the associated specifications by designated pragmas. The tool takes as input a source code file in which the SHVM to be analysed is represented by annotations. The product property, the variation point properties and the specifications of the public core methods are also provided by annotations. Figure 3 shows in the left column the annotation for the `@EnterProd` variation point, while the annotations for its `Keyboard` variant with core method `enterProd` are shown in the right column. PROMOVER fully automatically extracts the SHVM modules and the corresponding flow graphs from the annotated source code and performs the associated model checking tasks.

For evaluating our compositional verification approach, we considered the verification of the safety property explained in Example 5 for different versions of the trading system product line [24]. The product lines of cash desks were described as SHVMs with different hierarchical depths and different total numbers of modules. As a basis, we used the product line described in Example 1 and extended it by an optional coupon handling functionality within the `sale` method, and a variation point for accepting different card types as a hierarchical refinement of variant `Card`. For each product line, we compared the time required to verify all

Table 1. Evaluation Results

Product Line	Depth	# Modules	# Products	t_{ind} [s]	t_{comp} [s]
CD	1	7	9	79	9
CD/CH	1	9	18	177	10
CD/CT	2	15	27	278	11
CD/CH/CT	2	17	54	652	12

induced products individually with the time for compositional verification. The experiments were performed on a SUN SPARC machine⁴.

The results are summarized in Table 1 where CD denotes the product line of Example 1, CD/CH the version with coupon handling, CD/CT the version with different card types and CD/CH/CT the version with coupon handling and different card types. As can be observed from the table, the processing time t_{ind} for verifying every product individually grows dramatically when new modules and levels of hierarchy are added to the SHVM. This is easily explained by the analytical bounds presented in Section 2. In contrast, the growth of the processing time t_{comp} for compositional SHVM verification is insignificant, since the preprocessing and flow graph extraction is only performed once by PROMOVER for the complete SHVM. The experiment suggests that for large software products comprising many products, the compositional verification technique based on the SHVM representation of the product line increases efficiency of verification dramatically.

Scalability of our method comes at the price of having to provide specifications for variation points. This additional effort is justified for large systems that render infeasible the verification of the product line by verifying all its products individually. Also, the specifications only need to be written once and are later reused when the code has been changed, or for proving other global properties.

SHVMs do not allow to express that a variant requires or excludes another variant. Without these constraints, the set of products that can be derived from an SHVM is larger than with requires/excludes constraints. If a desired property can be shown for the larger set of products defined by an SHVM, the property immediately holds for the original product set defined by the hierarchical variability model. However, this leaves the possibility that not all products defined by an SHVM satisfy a property such that verification procedure fails, while the property is satisfied by the products defined by an hierarchical variability model containing variant constraints. In this case, an additional check of the excluded products would be required.

6 Related Work

The existing approaches to represent product line variability on the artifact level can be classified into three main directions [30]. First, annotative approaches

⁴ The focus of the evaluation is on comparing the times required for verification, and not on the total times themselves.

consider one model representing all products of a product line. Variant annotations, e.g., using UML stereotypes [31,9], presence conditions [6], or separate variability representations, such as orthogonal variability models [23], define which parts of the model have to be removed to derive the model of a concrete product. Second, compositional approaches [2,30,21,11] associate product fragments with product features which are composed for particular feature configurations. Third, transformational approaches, such as [14], represent variability by rules determining how modelling elements of a base model have to be replaced for a particular product model.

In this paper, we pursue an alternative approach to model the variability of a software product line by hierarchical variability modelling in SHVMs. Similar approaches are only pursued for modeling the variability of components contained in a software architecture. Plastic partial components [22] capture component variability by extending partially defined components with variation points and associated variants. However, variants cannot contain variable components, and thus the model is not fully hierarchical. In the Koala component model [29], component variability is defined by designated linguistic concepts, called diversity interfaces and switches, but these are fixed in a given component architecture.

Most approaches for algorithmic verification of behavioral properties of software product lines rely on an annotative model of the product line comprising all possible product variants in the same model. Existing model checking techniques are adapted to deal with optional behavior defined by variant annotations. For instance, in [8], modal transition systems are extended by variability operators from deontic logic. In [10], the process calculus CCS is extended with a variant operator to represent a family of processes. In [18], transitions of I/O-automata are related to variants. In [5], product families are modeled by transition systems where transitions are labelled with features, so that state reachability modulo a set of features can be computed.

These approaches do not scale for large product lines since the used annotative product line models easily get very large. To counter this, Blundell et al. [4] and Liu et al. [20] propose techniques for compositional verification of product features and are the only existing compositional verification techniques for product families in the literature so far. In these approaches, the behavior of a feature is represented by a state machine to which other features may attach in designated states (interface states or variation points). For a temporal property of a feature, constraints for these states are generated which have to be satisfied by composed features. However, the compositionality results are based on the applied notion of features and feature composition, while SHVMs provide a more flexible means to define product variability.

7 Conclusion

We present a novel hierarchical variability model for software product lines, in which the variability of products in terms of sets of public and private methods

is specified by defining common core methods and variation points at different hierarchical levels. The model allows to adapt a previously developed method and tool set for compositional verification of procedural programs such that the exponential blow-up required for verifying all products individually is avoided: The number of verification tasks resulting from our method is linear in the size of the variability model rather than in the number of products. This is achieved by the introduction of variation point specifications on which product properties are relativized, and the construction of maximal flow graphs that replace the specifications when model checking specifications on the next higher level of hierarchy. The class of properties that can be handled fully automatically is the class of control flow-based temporal safety properties, specifying illegal sequences of method calls. The input to our verification tool is the description of a product line in form of an annotated Java program defining the variability model and providing the necessary specifications.

Our first experiments with the tool show a dramatic gain in performance even for models with a low hierarchical depth. In future work, we plan to extend our hierarchical variability model with optional variants and constraints between variants in order to facilitate the direct verification of more expressive hierarchical variability models.

Acknowledgement. We thank Afshin Amighi for his help with flow graph extraction, and Björn Terelius for his help with obtaining the analytical bounds.

References

1. Apel, S., Janda, F., Trujillo, S., Kästner, C.: Model Superimposition in Software Product Lines. In: Paige, R.F. (ed.) ICMT 2009. LNCS, vol. 5563, pp. 4–19. Springer, Heidelberg (2009)
2. Batory, D., Sarvela, J., Rauschmayer, A.: Scaling Step-Wise Refinement. *IEEE Trans. Software Eng.* 30(6), 355–371 (2004)
3. Besson, F., Jensen, T., Le Métayer, D., Thorn, T.: Model checking security properties of control flow graphs. *J. of Computer Security* 9(3), 217–250 (2001)
4. Blundell, C., Fisler, K., Krishnamurthi, S., van Hentenryck, P.: Parameterized Interfaces for Open System Verification of Product Lines. In: *Automated Software Engineering (ASE 2004)*, pp. 258–267. IEEE, Los Alamitos (2004)
5. Classen, A., Heymans, P., Schobbens, P., Legay, A., Raskin, J.: Model Checking Lots of Systems: Efficient Verification of Temporal Properties in Software Product Lines. In: *International Conference on Software Engineering (ICSE 2010)*, pp. 335–344. IEEE, Los Alamitos (2010)
6. Czarnecki, K., Antkiewicz, M.: Mapping Features to Models: A Template Approach Based on Superimposed Variants. In: Glück, R., Lowry, M. (eds.) *GPCE 2005*. LNCS, vol. 3676, pp. 422–437. Springer, Heidelberg (2005)
7. Czarnecki, K., Eisenecker, U.W.: *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, Reading (2000)
8. Fantechi, A., Gnesi, S.: Formal Modeling for Product Families Engineering. In: *Software Product Line Conference (SPLC 2008)*, pp. 193–202. IEEE, Los Alamitos (2008)

9. Gomaa, H.: Designing Software Product Lines with UML. Addison Wesley, Reading (2004)
10. Gruler, A., Leucker, M., Scheidemann, K.: Modeling and model checking software product lines. In: Barthe, G., de Boer, F.S. (eds.) FMOODS 2008. LNCS, vol. 5051, pp. 113–131. Springer, Heidelberg (2008)
11. Grumberg, O., Long, D.: Model checking and modular verification. *ACM Transactions on Programming Languages and Systems* 16(3), 843–871 (1994)
12. Gurov, D., Huisman, M.: Reducing behavioural to structural properties of programs with procedures. In: Jones, N.D., Müller-Olm, M. (eds.) VMCAI 2009. LNCS, vol. 5403, pp. 136–150. Springer, Heidelberg (2009)
13. Gurov, D., Huisman, M., Sprenger, C.: Compositional verification of sequential programs with procedures. *Information and Computation* 206(7), 840–868 (2008)
14. Haugen, Ø., Møller-Pedersen, B., Oldevik, J., Olsen, G., Svendsen, A.: Adding Standardized Variability to Domain Specific Languages. In: Software Product Line Conference (SPLC 2008), pp. 139–148. IEEE, Los Alamitos (2008)
15. Huisman, M., Gurov, D.: CVPP: A tool set for compositional verification of control-flow safety properties. In: Beckert, B., Marché, C. (eds.) FoVeOOS 2010. LNCS, vol. 6528, pp. 107–121. Springer, Heidelberg (2011)
16. Kang, K., Lee, J., Donohoe, P.: Feature-Oriented Project Line Engineering. *IEEE Software* 19(4) (2002)
17. Kozen, D.: Results on the propositional μ -calculus. *Theoretical Computer Science* 27, 333–354 (1983)
18. Lauenroth, K., Pohl, K., Toehning, S.: Model checking of domain artifacts in product line engineering. In: Automated Software Engineering (ASE 2009), pp. 269–280. IEEE, Los Alamitos (2009)
19. Leavens, G., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., Müller, P., Kiniry, J., Chalin, P.: JML Reference Manual, Department of Computer Science, Iowa State University (February 2007), <http://www.jmlspecs.org>
20. Liu, J., Basu, S., Lutz, R.R.: Compositional model checking of software product lines using variation point obligations. *Autom. Softw. Eng.* 18(1), 39–76 (2011)
21. Noda, N., Kishi, T.: Aspect-Oriented Modeling for Variability Management. In: Software Product Line Conference (SPLC 2008), pp. 213–222. IEEE, Los Alamitos (2008)
22. Pérez, J., Díaz, J., Soria, C.C., Garbajosa, J.: Plastic Partial Components: A solution to support variability in architectural components. In: WICSA/ECSA, pp. 221–230 (2009)
23. Pohl, K., Böckle, G., van der Linden, F.: Software Product Line Engineering - Foundations, Principles, and Techniques. Springer, Heidelberg (2005)
24. Requirement Elicitation, Deliverable 5.1 of project FP7-231620 (HATS) (August 2009), <http://www.hats-project.eu>
25. Schwoon, S.: Model-Checking Pushdown Systems. PhD thesis, Technische Universität München (2002)
26. Soleimanifard, S., Gurov, D., Huisman, M.: Procedure-modular verification of control flow safety properties. In: Workshop on Formal Techniques for Java Programs, FTfJP 2010 (2010)
27. Soleimanifard, S., Gurov, D., Huisman, M.: Promover: Modular verification of temporal safety properties. In: Software Engineering and Formal Methods, SEFM 2011 (to appear, 2011)
28. Stirling, C.: Modal and Temporal Logics of Processes. Springer, Heidelberg (2001)

29. van Ommering, R.: Software reuse in product populations. *IEEE Trans. Software Eng.* 31(7), 537–550 (2005)
30. Völter, M., Groher, I.: Product Line Implementation using Aspect-Oriented and Model-Driven Software Development. In: *Software Product Line Conference (SPLC 2007)*, pp. 233–242. IEEE, Los Alamitos (2007)
31. Ziadi, T., Hëlouët, L., Jézéquel, J.-M.: Towards a UML Profile for Software Product Lines. In: van der Linden, F.J. (ed.) *PFE 2003. LNCS*, vol. 3014, pp. 129–139. Springer, Heidelberg (2004)

Variability Modelling in the ABS Language^{*}

Dave Clarke¹, Radu Muschevici¹, José Proença¹,
Ina Schaefer², and Rudolf Schlatte³

¹ IBBT-DistriNet, Katholieke Universiteit Leuven, Belgium

² University of Braunschweig, Germany

³ University of Oslo, Norway

Abstract. The HATS project aims at developing a model-centric methodology for the design, implementation and verification of highly configurable systems, such as software product lines, centred around the Abstract Behavioural Specification (ABS) modelling Language. This article describes the variability modelling features of the ABS Modelling framework. It consists of four languages, namely, μ TVL for describing feature models at a high level of abstraction, the Delta Modelling Language DML for describing variability of the ‘code’ base in terms of delta modules, the Product Line Configuration Language CL for linking feature models and delta modules together and the Product Selection Language PSL for describing a specific product to extract from a product line. Both formal semantics and examples of each language are presented.

1 Introduction

Software systems are central for the infrastructure of modern society. To justify the huge investment made to build such systems, they need to live for decades. This requires that the software is highly adaptable; software systems must support a high degree of variability to accommodate a range of requirements and deployment scenarios, and to allow these to change over time. A major challenge facing software construction is addressing high adaptability combined with trustworthiness. A limitation of current development practices is the missing rigour of models and property specification. Without a formal notation for distributed, component based systems, it is impossible to achieve automated consistency checking, security enforcement, generation of trustworthy code, etc. Furthermore, it does not suffice to simply extend current formal approaches.

Work done in the HATS project will make software product line engineering (SPLE) [30] into a more rigorous approach. SPLE addresses the development of software products sharing a number of commonalities, while differing in other aspects. Fig. 1 depicts the workflow in SPLE. Product variability can be expressed by *features*, which are user-visible product characteristics. The set of products is represented by a feature model [22,4], describing valid combinations of features. Given a set of software artefacts associated to these features, a final product is built by selecting the desired features and combining the artefacts.

^{*} This research is funded by the EU project FP7-231620 HATS: Highly Adaptable and Trustworthy Software using Formal Methods (<http://www.hats-project.eu>).

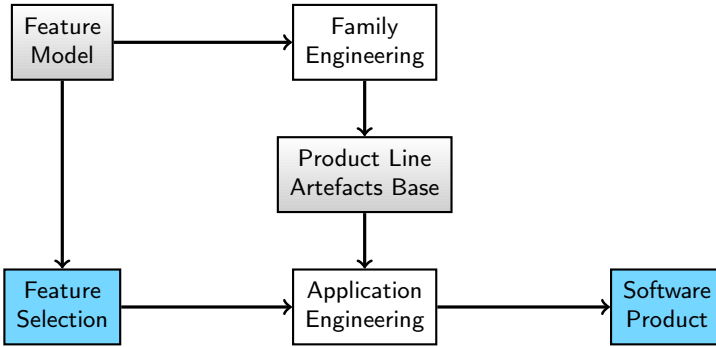


Fig. 1. Stages of product line development

The HATS project aims at developing a model-centric methodology for the design, implementation and verification of highly configurable systems, such as software product lines, that have high demands on dependability and trustworthiness. The HATS methodology is centred around the Abstract Behavioural Specification language (ABS) and its accompanying tool suite¹ that allows precise specification and analysis of the abstract behaviour and variability of highly configurable software systems. ABS is designed to fill the niche between design-oriented formalisms such as UML [27] and feature description language FDL [13], on one hand, and implementation-oriented formalisms such as Spec# [2] and JML [8], on the other hand. In this paper, we focus on the linguistic concepts of the ABS to represent anticipated system variability.

ABS [15] comprises a core language, called *Core ABS*, with specialised language extensions addressing system variability. Core ABS is a class-based, object-oriented language based on the active object concurrency model of Creol [21,6], which uses *asynchronous method calls* and *cooperative multi-tasking* between concurrent object groups of one or more ABS objects that share a computation resource; i.e., there can be at most one activity running inside the group.

The full ABS modelling framework extends Core ABS by four specialised languages to represent variability of Core ABS models. The *micro textual variability language* (μ TVL), based on TVL [7,11], expresses variability via feature models (Section 2). The *Delta Modelling Language* (DML), based on the concept of delta modelling [32], expresses the code-level variability of ABS models (Section 3). In delta modelling, a set of products is described by an initial core module, which is a Core ABS model, together with a set of product deltas specifying transformations to this core module (additions, removals, or modifications). The *Product Line Configuration Language* (CL) defines the relationship between the feature model and product deltas and thus forms the top-level specification of a product line of Core ABS models (Section 4). The *Product Selection Language* (PSL) represent the actual products by providing a selection of the product features

¹ <http://tools.hats-project.eu/>

and their attributes along with initialisation code for the product (Section 5). Fig. 2 depicts the relationship between these languages. The process of generating a software product from a software product line specification is explained in Section 6. Related work is presented in Section 7 and Section 8 concludes.

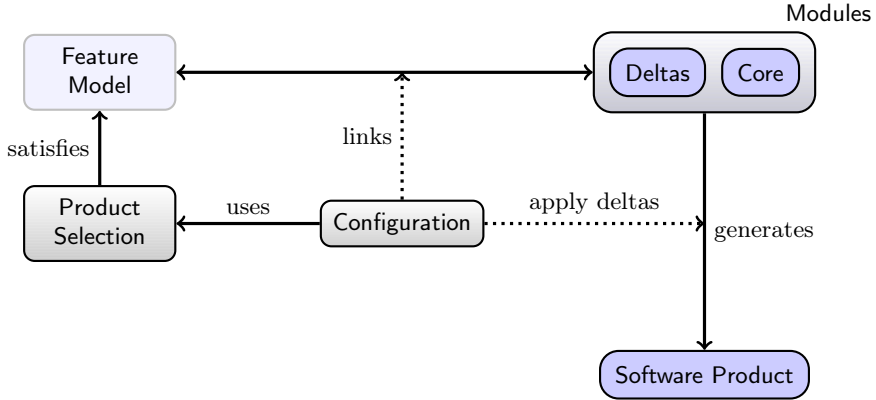


Fig. 2. Relationship Between Ingredients

2 Feature Modelling

This section introduces the μ TVL text-based feature modelling language, pronounced either *micro textual variability language* or simply *mu tee vee ell*, an extended subset of TVL [7,11]. TVL was developed at the University of Namur, Belgium, to serve as a reference language for specifying feature models. It is textual, as opposed to diagrammatic, and aims to be scalable, concise, modular, and comprehensive, and thus, serves as a suitable starting point for our purposes. A feature model is represented textually as a tree of nested features, each with a collection of boolean or integer attributes. Additional cross-tree dependencies can also be expressed in the feature model.

μ TVL is designed to be deliberately smaller than TVL in order to capture the essential feature modelling requirements and to simplify the manipulation of feature models. The simplification allows reducing a number of semantic constraints imposed by TVL to syntactic constraints. μ TVL enables a feature model with multiple roots (hence, multiple trees) to express orthogonal variability [30], which is useful for expressing application models and platform models in an orthogonal fashion (even in different files). Support for attributes of enumerated types have been dropped, but our tools support checking of satisfiability of integer attributes. Finally, in μ TVL features can only be extended (in *FeatureExtension* clauses) by adding new constraints, but not by introducing new features. Even though TVL syntax is used (with a few variations), the tools for μ TVL have been developed from scratch and integrated with the ABS language tool suite.

2.1 Concrete Syntax

The grammar of μ TVL is given in Fig. 3. Text in monospace denote terminal symbols. Assume the presence of two global sets: FID of feature names and AID of attribute names.

```

Model ::= (root FeatureDecl)* FeatureExtension*

FeatureDecl ::= FID [ { [Group] AttributeDecl* Constraint* } ]
FeatureExtension ::= extension FID { AttributeDecl* Constraint* }

Group ::= group Cardinality { [opt] FeatureDecl, ([opt] FeatureDecl)* }
Cardinality ::= allof | oneof | [n1 .. *] | [n1 .. n2]
AttributeDecl ::= Int AID ; | Int AID in [ Limit .. Limit ] ; | Bool AID ;
Limit ::= n | *

Constraint ::= Expr ; | ifin: Expr ; | ifout: Expr ;
                | require: FID ; | exclude: FID ;
Expr ::= True | False | n | FID | AID | FID.AID
                | UnOp Expr | Expr BinOp Expr | ( Expr )
UnOp ::= ! | -
BinOp ::= || | && | -> | <-> | == | != | > | < | >= | <= | + | - | * | / | %
    
```

Fig. 3. Grammar of μ TVL; n ranges over integers

Attributes and values in μ TVL range either over integers or booleans. The *Model* clause specifies a number of ‘orthogonal’ root feature models along with a number of extensions that specify additional constraints, typically cross-tree dependencies. The *FeatureDecl* clause specifies the details of a given feature, firstly by giving it a name (FID), followed by a number of possibly optional sub-features, the feature’s attributes and any relevant constraints. The *FeatureExtension* clause specifies additional constraints and attributes for a feature. This is particularly useful for specifying constraints that do not fit into the tree structure given by the root feature model. The *Cardinality* clause describes the number of elements of a group that may appear in a result. The *AttributeDecl* clause specifies the declaration of both integer (bounded or unbounded) and boolean attributes of features.

The *Constraint* clause specifies constraints on the presence of features and on attributes. An **ifin** constraint is only applicable if the current feature is selected. Similarly, an **ifout** constraint is only applicable if the current feature is not selected. A **require** clause specifies that the current feature requires some other feature, whereas **exclude** expresses the mutual incompatibility between the current feature and some other feature. The *Expr* clause expresses a boolean constraint over the presence of features and attributes, using standard boolean and arithmetic operators. Features are referred to by identity (FID). Attributes are referred to either using an unqualified name (AID), for in scope attributes, or using a qualified name (FID.AID) for attributes of other features.

Example 1. The following is a feature model of a *multi-lingual Hello World* product line, which describes software that can output “Hello World” in multiple languages some number of times.

```

root MultiLingualHelloWorld {
  group allof {
    Language {
      group oneof { English, Dutch, German }
    },
    opt Repeat {
      Int times in [0..1000];
      ifin: times > 0;
    } } }

```

```

extension English {
  ifin: Repeat ->
    (Repeat.times >= 2 &&
     Repeat.times <= 5);
}

```

The *multi-lingual Hello World* product line in the example above has two main features, *Language* and *Repeat*, under the root feature and joined with the **allof** combinator. The *Language* feature requires one out of three possible features: *English*, *Dutch*, or *German*. The *Repeat* feature is optional, it has no associated sub-features, and it has an attribute *times* which ranges between 0 and 1000, with an added condition that it must be strictly greater than 0. In this example an extension for the *English* feature is given. When the *English* and the *Repeat* features are present, the attribute *times* must be between 2 and 5, inclusive.

2.2 Abstract Syntax

The abstract syntax tree for μ TVL programs is presented in Fig. 4, where $f \in \text{FID}$, $a \in \text{AID}$, and $n \in \text{Int}$. The translation from the concrete tree to the abstract tree is straightforward and hence omitted. Local attribute names are expanded to fully qualified names. Bounds are placed on all integer attributes. The semantics of μ TVL is given as the solutions of the integer constraints defined inductively on the abstract syntax of feature models.

2.3 Semantics

The semantics of a feature model in μ TVL are defined by translation into constraints over integers whose solutions correspond to valid feature and attribute selections. Boolean variables are treated as integers in the standard manner: 0 corresponds to false, and 1 to true. The function $\llbracket \cdot \rrbracket$ encoding feature model M as an integer constraint is given in Fig. 5. The notation \bar{x} represents a sequence of elements $x_1 \cdots x_n$. Within the context of a given feature f , function $\llbracket \cdot \rrbracket_f$ translates constraints relative to that feature. In the translation, f^\dagger is a unique name based on name f . If f is an optional feature, f^\dagger can freely be set to 1 to count the optional feature, even when f is absent. For example, when dealing with an **allof** constraint, it is required that all children are present; some may however be optional, so as far as the **allof** constraint is concerned, optional children are counted, though the corresponding features may not be included. Expressions e are encoded into constraints, denoted ϕ_e . Their encoding is straightforward and therefore omitted (see [11]). Boolean operations are mapped to a conjunctive

$M ::= F^*$	feature model	$C ::= e \mid \text{ifin } e \mid \text{ifout } e \mid$	
$F ::= f [G] A^* C^*$	feature (extension)	$\mid \text{require } f \mid \text{exclude } f$	constraint
$G ::= c N^*$	group	$lt ::= \text{true} \mid \text{false}$	
$N ::= \text{opt } F \mid \text{mand } F$	feature node	$\mid n \mid f \mid f.a$	literal or variable
$c ::= \text{allof} \mid \text{min } n$		$U ::= \text{neg} \mid \text{not}$	unary operator
$\mid \text{rng } n \ n$	cardinality	$B ::= \text{or} \mid \text{and} \mid \text{implies}$	
$A ::= f.a T$	attribute declaration	$\mid \text{equiv} \mid \text{eq} \mid \text{neq}$	
$T ::= \text{bool} \mid \text{int } L \ L$	type and domain	$\mid \text{lt} \mid \text{gt} \mid \text{lteq}$	
$L ::= * \mid n$	domain limit	$\mid \text{gteq} \mid \text{plus} \mid \text{minus}$	
$e ::= lt \mid U e \mid B e e$	expression	$\mid \text{mult} \mid \text{div} \mid \text{mod}$	binary operator

Fig. 4. Abstract syntax of μTVL

set of integer operations over the values 0 and 1 where, for example, $a \rightarrow b$ is a shorthand for $a \leq b$. Finally, we assume a lower bound MIN and an upper bound MAX on the values of integer variables.

Given a feature model FM in μTVL , the set of solutions of the integer constraints $\llbracket FM \rrbracket$ provides our semantics for FM . Such a solution will specify values for all attributes even when the corresponding feature is not selected. Such assignments should have no effect.

The semantics also enforce that each feature is selected either zero or one times, in spite of cardinality conditions which may appear to allow more instances of a feature. Cardinality conditions specify the number of selected sub-features from a group. Note that optional features can only appear under the **allof** cardinality; otherwise there would be a fragile interaction between cardinality conditions and optional features [5].

Example 2. Below is the encoding into integer constraints of the Hello World feature model introduced in Example 1.

```

0 ≤ MultiLingualHelloWorld ≤ 1 ∧
Language → MultiLingualHelloWorld ∧ Repeat† → MultiLingualHelloWorld ∧
Language + Repeat† = 2 ∧
0 ≤ Language ≤ 1 ∧
English → Language ∧ Dutch → Language ∧ German → Language ∧
1 ≤ English + Dutch + German ≤ 1 ∧
0 ≤ English ≤ 1 ∧ 0 ≤ Dutch ≤ 1 ∧ 0 ≤ German ≤ 1 ∧
0 ≤ Repeat† ≤ 1 ∧
Repeat → Repeat† ∧
0 ≤ Repeat ≤ 1 ∧ 0 ≤ Repeat.times ≤ 1000 ∧ Repeat.times > 0 ∧
English → (Repeat → (Repeat.times ≥ 2 ∧ Repeat.times ≤ 5)).
    
```

Every declaration of a new feature or attribute x is converted into a constraint of type $min \leq x \leq max$, and, in the case of booleans and feature names, $min = 0$ and $max = 1$. The tree structure of the feature model is captured by implications between the children and their parents, as shown in the second line of Example 2. The optional feature **Repeat** is split into two variables: **Repeat** and **Repeat[†]**. The latter is used only to address the cardinality of the parent **MultiLingualHelloWorld**, and they are connected by the implication

$$\begin{aligned}
\llbracket \overline{F} \rrbracket &= \bigwedge_{x \in \overline{F}} \llbracket x \rrbracket \\
\llbracket f [G] \overline{A} \overline{C} \rrbracket &= (0 \leq f \leq 1) \wedge \llbracket [G] \rrbracket_f \wedge \llbracket \overline{A} \rrbracket \wedge \llbracket \overline{C} \rrbracket_f \\
\llbracket \text{all of } \overline{N} \rrbracket_f &= \text{tree}(f, \overline{N}) \wedge \sum \overline{N} = \# \overline{N} \wedge \llbracket \overline{N} \rrbracket \\
\llbracket (\min n) \overline{N} \rrbracket_f &= \text{tree}(f, \overline{N}) \wedge n \leq \sum \overline{N} \wedge \llbracket \overline{N} \rrbracket \\
\llbracket (\text{rng } n_1 \ n_2) \overline{N} \rrbracket_f &= \text{tree}(f, \overline{N}) \wedge n_1 \leq \sum \overline{N} \leq n_2 \wedge \llbracket \overline{N} \rrbracket \\
\llbracket \text{opt } (f [G] \overline{A} \overline{C}) \rrbracket &= f \rightarrow f^\dagger \wedge \llbracket f [G] \overline{A} \overline{C} \rrbracket \\
\llbracket \text{mand } F \rrbracket &= \llbracket F \rrbracket \\
\llbracket f.a \text{ int } L_1 \ L_2 \rrbracket &= \text{val}_{\min}(L_1) \leq f.a \wedge \\
&\quad f.a \leq \text{val}_{\max}(L_2) \\
\llbracket f.a \text{ bool} \rrbracket &= 0 \leq f.a \leq 1 \\
\llbracket e \rrbracket &= \phi_e \\
\llbracket \text{ifin } e \rrbracket_f &= f \rightarrow \llbracket e \rrbracket \\
\llbracket \text{ifout } e \rrbracket_f &= \neg f \rightarrow \llbracket e \rrbracket \\
\llbracket \text{require } f' \rrbracket_f &= f \rightarrow f' \\
\llbracket \text{exclude } f' \rrbracket_f &= \neg(f \wedge f') \\
\llbracket [X] \rrbracket &= \begin{cases} \llbracket X \rrbracket & \text{if } X \text{ is present} \\ \text{true} & \text{otherwise} \end{cases} & \text{feat}(\text{opt}(f _ _ _)) = f^\dagger \\
\#(N_1 \cdots N_n) &= n & \text{feat}(\text{mand}(f _ _ _)) = f \\
\sum(N_1 \cdots N_n) &= \text{feat}(N_1) + \cdots + \text{feat}(N_n) & \text{val}_x(n) = n \\
\text{tree}(f, N_1 \cdots N_n) &= \bigwedge_{1 \leq i \leq n} \text{feat}(N_i) \rightarrow f & \text{val}_{\min}(\ast) = \text{MIN} \\
& & \text{val}_{\max}(\ast) = \text{MAX}
\end{aligned}$$

Fig. 5. Semantics of μ TVL

Repeat \rightarrow **Repeat**[†], similar to how child features are related to their parent. Cardinalities are encoded as constraints that add the 0-1-integer value of the feature variables and check whether they belong to a specific domain, as shown in the third and seventh line of the example. Constraints over attributes are simply interpreted as integer constraints.

3 Delta Modelling

Delta-oriented programming was introduced by Schaefer et al. [32,34,33] as a novel programming language approach for software-based product lines, and as an direct alternative to feature-oriented programming [3]. Both approaches aim at automatically generating software products for a given feature selection by providing a flexible and modular technique to build different products that share common code. In feature-oriented programming, software modules are associated to features, and product generation consists of composing the modules for a feature selection. In delta-oriented programming [32], *application conditions* over the set of features and their attributes, are associated with modules of program modifications (add, remove or modify code), called delta modules. The collection of applicable delta modules is given by the application conditions that are true for a particular feature and attribute selection. By not associating the delta modules directly with features, a degree of flexibility is obtained, resulting in better reuse of

code and the ability to resolve conflicts caused by deltas modifying the code base in incompatible ways [10]. The flexibility offers benefits for managing the evolution of product lines, by allowing versions to be implemented using software deltas.

The implementation of a software product line in delta-oriented programming [32] is divided into a *core module* and a set of *delta modules*. The core module consists of the classes that implement a complete product of the corresponding product line. Delta modules describe how to change the core module to obtain new products. The choice of which delta modules to apply is based on the selection of desired features for the final product. Schaefer et al. described and implemented delta-oriented programming for *Java* [32], introducing the programming language DELTAJAVA. This language has strongly influenced our design, though we further separate deltas from features by moving application conditions out of deltas and into a product line configuration language, as pursued in [34,33]. Delta modelling is included in the ABS language to implement variability at the source code level of abstraction.

3.1 Syntax

Figure 6 specifies the ABS syntax related to delta modelling. Nonterminals written in purple (gray) refer to core ABS symbols, whose intended meaning should be immediate.

```

DeltaDecl ::= delta TypeId [DeltaParams] { ClassOrIfaceModifier* }
ClassOrIfaceModifier ::= adds ClassDecl
                        | modifies class TypeName ImplModifier* { Modifier* }
                        | removes class TypeName ;
                        | adds InterfaceDecl
                        | modifies interface TypeName ImplModifier* { Modifier* }
                        | removes interface TypeName ;

ImplModifier ::= adds TypeName
                | removes TypeName

Modifier ::= adds FieldDecl
             | removes FieldDecl
             | adds MethDecl
             | modifies MethDecl
             | removes MethSig

DeltaParams ::= (DeltaParam (, DeltaParams)* )
DeltaParam ::= Identifier HasCondition*
              | Type Identifier

HasCondition ::= hasField FieldDecl
                | hasMethod MethSig
                | hasInterface TypeName

```

Fig. 6. ABS Grammar: Delta Modules

The *DeltaDecl* clause specifies the syntax of delta modules, consisting of an unique identifier, a list of parameters and a body containing a sequence of class and interface modifiers. The *ClassOrIfaceModifier* clause describes the syntax of modifications at the level of classes and interfaces. Such a modification can add a class or interface declaration, modify an existing class or interface, or remove a class or interface. The *ImplModifiers* clause describes how to modify the interfaces a class implements or an interface extends, either by adding new or removing existing interfaces.

The *Modifier* clause specifies the modifications that can occur within a class or interface body. These include (where relevant) adding and removing fields and method signatures (from interfaces), and modifying methods, which amounts to replacing a method with a new one, but enabling the original method to be called using the **original** keyword. The aim of **original** is to enable the method being replaced to be called from the delta module that replaces it. This is implemented by renaming the original method, and replacing the call via keyword **original** with a call to the renamed method. The semantics of calling **original()** as shown in the above example are essentially the same as **Super()** from feature-oriented programming [3], and **proceed** from context-oriented programming [19], and similar to ordinary **super** calls in standard object-oriented languages, as well as **around** advice from aspect-oriented programming [23], except without quantification.

In contrast to deltas presented in the literature [32,34,33], delta modules in the HATS ABS language can be parameterised both by attribute values, which ultimately flow from the feature model selection, and by class names, to enable the application of a single delta module in more than one circumstance. Finally, the *HasCondition* describes constraints on class arguments to which a delta may be applied. These constraints consist of descriptions of the methods and fields such a class implements and any interfaces it is expected to have.

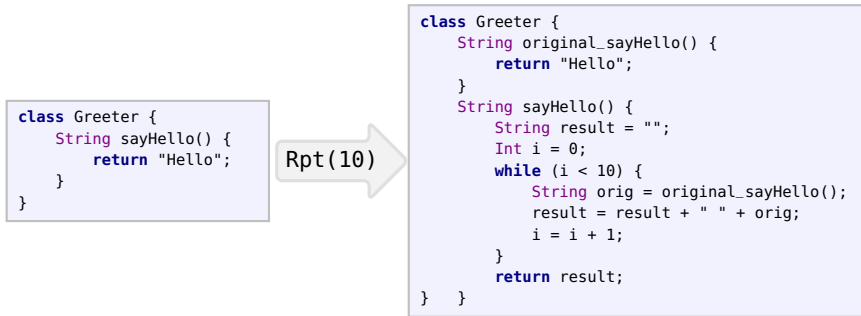
Example 3. Following is the implementation of the Hello World product line with the feature model shown in Example 1. Delta modules specify the variable behaviour.

```
interface Greeting {
    String sayHello();
}
class Greeter implements Greeting {
    String sayHello() {
        return "Hello world";
    }
}
class Application {
    String s = "";
    Unit run() {
        Greeting bob;
        bob = new Greeter();
        s = bob.sayHello();
    }
}
delta N1 {
    modifies Greeter {
        modifies String sayHello() {
            return "Hallo wereld";
        }
    }
}
```

```
delta De {
    modifies Greeter {
        modifies String sayHello() {
            return "Hallo Welt";
        }
    }
}
delta Rpt (Int times) {
    modifies Greeter {
        modifies String sayHello() {
            String result = "";
            Int i = 0;
            while (i < times) {
                String orig = original();
                result = result + " " + orig;
                i = i + 1;
            }
            return result;
        }
    }
}
```

In the example above the interface `Greeting` and the classes `Greeter` and `Application` form the core module of the implementation, written in the core ABS language. There are three delta modules: `Nl`, `De`, and `Rpt`. The delta module `De` has a single class modifier for `Greeter`, which in turn has a single method modifier. This method modifier replaces the method `sayHello` to return the German text “Hallo Welt”. The delta module `Rpt` has a single parameter for the number of times that the greeting should be repeated. It replaces the method `sayHello()` inside `Greeter` with new ABS code, allowing the original method to be called via `original()`.

Example 4. The following diagram illustrates both the use of parameters and of the `original` keyword. A parameterised delta, such as `Rpt`, must have its arguments provided before it can be applied. The arguments are substituted into the body of the delta module prior to application.



3.2 Formal Semantics

Applying a delta module Δ to a core ABS program P yields a new core ABS program. Thus a product is constructed by successively applying delta modules, one at a time, to a core module. This section presents a formal semantics of delta modules based on the more abstract presentation of Clarke et al. [10]. That work also describes the composition of delta modules with each other, which is essential for reasoning about conflicting delta modules, but this feature is elided from the current presentation. ABS programs, classes and delta modules will be represented in terms of finite maps from identifiers to the corresponding contents of the program, class, or delta module, in order to more cleanly present the semantics. The semantics only describes the modifications of methods; dealing with fields and so forth is a straightforward extension. Parameters are omitted. These will be treated when dealing with configurations in Section 4.

Let *Identifier* be the set of identifiers, let *MethBody* be the set of method bodies, including the parameter and return types, and let *MethBodyWrap* be the set of method bodies with an explicit call to `original`. In the following domains, `Replace`, `Update`, and `Remove` are used to tag the various branches of sum data types. Finally, let *Error* denote that an error has occurred. Errors occur if one attempts to wrap a method that is not present. Other irregularities,

such as attempting to update a class that is not present, can be given a sensible semantics, so long as no wrapping occurs.

$$\begin{aligned}
\text{Program} &= \text{Identifier} \rightarrow \text{ClassBody} \\
\text{ClassBody} &= \text{Identifier} \rightarrow \text{MethBody} \\
\text{Delta} &= \text{Identifier} \rightarrow \text{DeltaBody} \\
\text{DeltaBody} &= \text{Replace} (\text{Identifier} \rightarrow \text{MethBody}) \\
&\quad \uplus \text{Update} (\text{Identifier} \rightarrow (\text{MethBody} \uplus \text{MethBodyWrap} \uplus \text{Remove})) \\
&\quad \uplus \text{Remove}
\end{aligned}$$

A program is a map from class names to classes, which themselves are collections of named method bodies. A delta module is a map from class names to delta bodies, which consist of three different types of modification: **Replace** either adds or replaces the class with the specified contents; **Update** modifies a class in place, where the three elements within an update clause correspond to replacing a method with a new body from *MethBody*, wrapping the method with a body from *WrapMethBody* or removing the method; and finally, **Remove** denotes the removal of the class.

Notation 1. Let $f : X \rightarrow Y$ denote a partial function from X to Y . If $f(x)$ is undefined for $x \in X$, write $f(x) = \perp$, where $\perp \notin Y$. For set A , let A_\perp denote $A \cup \{\perp\}$, where $\perp \notin A$. We freely shift between partial functions $X \rightarrow Y$ and functions $X \rightarrow Y_\perp$. If $\odot : A_\perp \times B_\perp \rightarrow C_\perp$, define the lifting of \odot to partial functions over index set I as

$$\begin{aligned}
-\overline{\odot}- : (I \rightarrow A) \times (I \rightarrow B) &\rightarrow (I \rightarrow C) \\
(f \overline{\odot} g)(i) &= f(i) \odot g(i), \quad \text{where } i \in I
\end{aligned}$$

Given class update $f : \text{Identifier} \rightarrow (\text{MethBody} \uplus \text{MethBodyWrap} \uplus \text{Remove})$, define function $f^* : \text{Identifier} \rightarrow (\text{MethBody} \uplus \text{Error})$ as follows. For $i \in \text{Identifier}$:

$$f^*(i) = \begin{cases} \perp & \text{if } f(i) = \text{Remove} \\ f(i) & \text{if } f(i) \in \text{MethBody} \\ \text{Error} & \text{if } f(i) \in \text{MethBodyWrap}. \end{cases}$$

Notation 2. In the following definition, the notation $w[\]$ denotes a wrapper method from *MethBodyWrap*, where the hole $[\]$ denotes that the original method is unknown. Notation $w[b]$ denotes the wrapping of method body b with wrapper w , thus the **original** call can be successfully bound. The resulting method $w[b]$ is considered to be an element of *MethBody*.

Definition 1 (Delta module application). The application of a delta module to a program is specified by the following functions:

$$\begin{aligned}
\text{apply} &: \text{Delta} \times \text{Program} \rightarrow \text{Program} \\
\text{apply}(d, p) &= d \overline{\odot}_c p
\end{aligned}$$

$$\begin{array}{l}
 \text{where} \quad - \odot_c - : \mathit{DeltaBody}_\perp \times \mathit{ClassBody}_\perp \rightarrow \mathit{ClassBody}_\perp \\
 \quad \quad \quad \perp \odot_c x = x \\
 \quad \quad \quad (\mathit{Replace} \ g) \odot_c _ = g \qquad (\mathit{Update} \ f) \odot_c \perp = f^* \\
 \quad \quad \quad \mathit{Remove} \odot_c _ = \perp \qquad (\mathit{Update} \ f) \odot_c h = f \overline{\odot_m} h \\
 \\
 \text{and} \quad - \odot_m - : (\mathit{MethBody} \uplus \mathit{MethBodyWrap} \uplus \mathit{Remove})_\perp \times \mathit{MethBody}_\perp \\
 \quad \quad \quad \rightarrow (\mathit{MethBody} \uplus \mathit{Error})_\perp \\
 \quad \quad \quad \perp \odot_m x = x \\
 \quad \quad \quad w[\] \odot_m b = w[b] \qquad m \odot_m _ = m \\
 \quad \quad \quad \mathit{Remove} \odot_m _ = \perp \qquad w[\] \odot_m \perp = \mathit{Error}
 \end{array}$$

where $m \in \mathit{MethBody}$ and $w[\] \in \mathit{MethBodyWrap}$.

Notation 3. If $m \in \mathit{Identifier}$ then $w[m]$ denotes the wrapper with each call to **original** replaced by a call to m .

In our implementation the method body is not inlined. Instead, if the resulting class C has an element $m \mapsto w[b] \in C$, the following post-processing steps are performed before applying another delta module:

1. generate a fresh method name $m' \notin C$,
2. remove $m \mapsto w[b]$ from C , and
3. add $m \mapsto w[m']$ and $m' \mapsto b$ to C .

Example 4 illustrated this process with concrete code. The modified method in that example is `sayHello()`. Before replacing the method, it was renamed to a fresh name such as `original_sayHello()`. The new method was then added to the class, with its body modified so that `original` is replaced by the renamed method's name `original_sayHello`. The stipulation that the method name is fresh is required in the case that multiple delta modules are applied to the same class, each wrapping the same method. In such a case, the first renaming would result in method `original_sayHello`, for example, the second in name `original_sayHello2`, and so forth.

4 Product Line Configuration

This section describes the product line configuration language CL which links feature models specified in μTVL (Section 2) with delta modules (Section 3), to specify the variability in a product line. This approach is similar to the product line specification proposed in delta-oriented programming [34,33].

A product line configuration consists of a set of features assumed to exist and a set of *delta clauses*. Each delta clause specifies a delta and the conditions required for its application, propositional formulas over the set of known features and attributes called *application conditions*, and a partial ordering relation with respect to other deltas. When the propositional formula holds for a given product, the delta is said to be active. The partial order states which deltas, when active, should be applied before the current delta.

```

Configuration ::= productline TypeId { Features ; Deltas }
    Features ::= features FID ( , FID ) *
DeltaClauses ::= DeltaClause ( , DeltaClause ) *
DeltaClause ::= delta DeltaSpec [AfterCondition] [ApplicationCondition] ;
    DeltaSpec ::= TypeName [( DeltaArgs )]
    DeltaArgs ::= DeltaArg ( , DeltaArg ) *
    DeltaArg ::= FID | FID.AID | DataExp

    AfterCondition ::= after TypeName ( , Name ) *
    ApplicationCondition ::= when Expr

```

Fig. 7. Product Line Configuration Grammar

4.1 Syntax

The syntax of the product line configuration language is given in Fig. 7. The *Configuration* clause specifies the name of the product line, the set of features it implements, and the set of delta modules used to implement those features. The feature names are included so that certain simple self-consistency checks can be performed. The *DeltaClause* clause is used to specify each delta module, linking it to the feature model. Each *DeltaClause* has a *DeltaSpec*, specifying its name and its parameters, an *AfterCondition*, specifying the delta modules that the current delta must be applied after, and an *ApplicationCondition*, specifying an arbitrary predicate over the feature and attribute names (see Fig. 3) that describes when the given delta module is included in the product line.

Example 5. The Hello World product line is configured, connecting the features and attributes defined in the feature model to delta modules.

```

productline MultiLingualHelloWorld {
    features English, German, Dutch, Repeat;

    delta Rpt(Repeat.times) after De, NL when Repeat;
    delta De when German;
    delta NL when Dutch;
}

```

The example above first names the set of features from the feature model in Example 1 used to configure this product line. The **delta** clauses link each delta module to the feature model through an application condition (**when** clause); in this case, a delta module is applied simply when the specified feature is selected (e.g. “De **when** German”). There is no delta module corresponding to the feature English, as the core module provides support for the English language by default. In addition, Rpt has to be applied **after** De and NL. Rpt’s argument is Repeat.times, the times attribute feature Repeat; its value (defined by product selection, see Section 5) is propagated to the Rpt delta.

4.2 Semantics

A CL script specifies how the feature model relates to the delta modules that are to be applied to the core module. It does so by specifying the parameters and application conditions for each delta module, and an ordering on the deltas.

Each delta module referred to in a configuration file is modelled by an element of the following type:

$$\mathit{Delta} \times \mathit{Params} \times \mathit{AppCondition}$$

where Delta is the semantic domain of delta module bodies, defined in Section 3.2,

$$\mathit{Params} = \mathit{Var} \rightarrow \mathit{FID} \uplus (\mathit{FID} \times \mathit{AID}) \uplus \mathit{Int}$$

models the substitution of actual parameters, which may be attributes or constants, defined in the CL script with the formal parameters of the corresponding delta module, and $\mathit{AppCondition}$ is the syntactic category of application conditions. Class parameters to delta modules are not modelled.

A configuration script can be modelled as a partial order over the declared delta modules (with their parameters and application conditions), where the partial order is determined by the reflexive, transitive closure of the **after** clauses. This is given by the following domain, where $\mathit{PO}(-)$ denotes the collection of all partial orders over a given set.

$$\mathit{Config} = \mathit{PO}(\mathit{Delta} \times \mathit{Params} \times \mathit{AppCondition})$$

The semantics of a configuration script $\mathit{conf} \in \mathit{Config}$ is a function of type

$$\llbracket \mathit{conf} \rrbracket_- : \mathit{ProductSelection} \rightarrow \mathcal{P}(\mathit{Delta}^*)$$

which maps a product selection—the interpretation of a PSL script (see Section 5.2)—to the delta modules to apply, in the order they should be applied. Note that many orders may exist if the **after**-order is underspecified. A product selection is an assignment from feature names to true or false (1 or 0) and from attributes to values, given by the domain $\mathit{ProductSelection}$:

$$\mathit{ProductSelection} = (\mathit{FID} \uplus (\mathit{FID} \times \mathit{AID})) \rightarrow \mathit{Int}$$

We now develop the ingredients making up function $\llbracket \mathit{conf} \rrbracket_-$.

Firstly, assume that a notion of substitution exists for delta modules, respecting the scoping of variables, to replace parameters with appropriate values:

$$\begin{aligned} \mathit{Subst} &= \mathit{Var} \rightarrow \mathit{Int} \\ \mathit{applySubst} &: \mathit{Subst} \times \mathit{Delta} \rightarrow \mathit{Delta} \end{aligned}$$

Next, we define the composition of the parameter specifications of delta modules with a product selection, giving a mapping from formal parameters of delta

modules to values (Int), which will be used to refine the delta modules with the configuration parameters specifying in the product selection:

$$\begin{aligned} \circ : \text{ProductSelection} \times \text{Params} &\rightarrow \text{Subst} \\ \sigma \circ p &= \{v \mapsto x\sigma \mid v \mapsto x \in p\} \\ \text{where } x\sigma &= \begin{cases} v & \text{if } x \in \text{FID} \uplus (\text{FID} \times \text{AID}) \text{ and } x \mapsto v \in \sigma \\ x & \text{if } x \in \text{Int} \end{cases} \end{aligned}$$

Now the function taking a product selection $\sigma \in \text{ProductSelection}$ and giving the collection of delta modules to apply is computing as the composition of the following steps:

1. Select applicable deltas by applying $\text{select}_\sigma : \text{Config} \rightarrow \text{PO}(\text{Delta} \times \text{Params})$

$$\text{select}_\sigma(D, \prec) = (D', \prec|_{D'}),$$

where $D' = \{(d, p) \mid (d, p, \phi) \in D, \sigma \models \phi\}$ and $\prec|_{D'}$ is \prec restricted to D' , and $\models \subseteq \text{ProductSelection} \times \text{AppCondition}$ is the satisfaction relation.

2. Specialise deltas using the function $\text{specialise}_\sigma : \text{ProductSelection} \times \text{PO}(\text{Delta} \times \text{Params}) \rightarrow \text{PO}(\text{Delta})$

$$\text{specialise}_\sigma(D, \prec) = (D', \prec|_{D'}), \text{ where } D' = \{\text{applySubst}(\sigma \circ p, d) \mid (d, P) \in D\}.$$

3. Order deltas using the function $\text{order} : \text{PO}(\text{Delta}) \rightarrow \mathcal{P}(\text{Delta}^*)$

$$\text{order}((D, \prec)) = \{[d_1, \dots, d_n] \mid d_1, \dots, d_n \text{ is a linear extension of } (D, \prec)\}.$$

Finally, the semantics of a CL script can be interpreted as a function

$$\begin{aligned} \llbracket _ \rrbracket_\sigma &: \text{Config} \times \text{ProductSelection} \rightarrow \mathcal{P}(\text{Delta}^*) \\ \llbracket \text{conf} \rrbracket_\sigma &= \text{order}(\text{specialise}_\sigma(\text{select}_\sigma(\text{conf}))). \end{aligned}$$

Note that this process may be ambiguous when multiple orderings of delta modules are possible. This should be resolved either by adding more elements to the ‘**after**’ order or by introducing conflict-resolving deltas [10].

5 Product Selection

A product selection needed to generate a product from a product line is specified using the *product selection language* (PSL). A product selection states which features are to be included in the product and by sets attributes of those features to concrete values. In addition, some core ABS code is provided to initialise the selected product. As depicted in Fig. 2, a product selection is checked against a μTVL feature model for validity. It is then used by the configuration file to guide the selection and application of deltas during the generation of the final software product.

5.1 Syntax

Fig. 8 specifies the grammar of the ABS product selection language. The *Selection* clause specifies a product by giving it a name, by stating the features and optional attribute assignments that are included in that product, and by specifying an initialisation block. An initialisation block can be any core ABS block, but typically will be a simple call to some already present *main* method. Initialisation blocks are specified in the product selection language to enable product lines with multiple entry points to start execution.

Example 6. Products of the Hello World product line are product selections.

```
// basic product with no deltas
product P1 (English) {
  new Application();
}

// apply delta De
product P2 (German) {
  new Application();
}
```

```
// apply deltas De and Repeat
product P3 (German, Repeat{times=10}) {
  new Application();
}
// apply deltas En and Repeat, but it
// should be refused because "times > 5"
product P4 (English, Repeat{times=6}) {
  new Application();
}
```

In the example above we specify four products: P1, P2, P3, and P4. In the case of the product P1, the parameter **English** means the product consists of this feature and of the features implied by the constraints over the feature model. In this case the implied features are **Language** and the root **MultiLingualHelloWorld**, according to the model in Example 1. In P3 and P4 the parameters also include attribute values, in these cases assigning a value to the attribute **times** from the feature **Repeat**. The block of ABS code associated to each product provides its initialisation code. Every product in our example instantiates an **Application** object and executes its **run** method.

5.2 Semantics

There are two components of interest in a PSL product selection such as

```
product P (Feature1 {attribute1_1 = value1_1, ...},
          Feature2 {attribute2_1 = value2_1, ...}, ...)
{ InitBlock }
```

$$\textit{Selection} ::= \textit{product } \textit{TypeId} (\textit{FeatureSpecs}) \{ \textit{InitBlock} \}$$

$$\textit{FeatureSpecs} ::= \textit{FeatureSpec} (, \textit{FeatureSpec})^*$$

$$\textit{FeatureSpec} ::= \textit{FID} [\textit{AttributeAssignments}]$$

$$\textit{AttributeAssignments} ::= \{ \textit{AttributeAssignment} (, \textit{AttributeAssignment})^* \}$$

$$\textit{AttributeAssignment} ::= \textit{AID} = \textit{Literal}$$

$$\textit{InitBlock} ::= \textit{Block}$$

Fig. 8. PSL Grammar

- An assignment $\sigma \in \text{ProductSelection}$ defined as follows:
 - for each `Featurei`, $\sigma(\text{Feature}_i) = 1$.
 - for each `attributei,j = valuei,j` clause in `Featurei`,
 $\sigma(\text{Feature}_i.\text{attribute}_{i,j}) = \text{value}_{i,j}$.
- The initialisation block.

The assignment is not complete as it does not specify the values for unselected or implicitly-selected features. An example of an implicitly-selected feature occurs when a leaf feature is selected, requiring that its ancestors in the tree need to be selected too. In addition, the variable f^\dagger introduced to count optional feature f is set to 1. Finally, values of attributes for unselected features are set to some arbitrary value so that the all variables appearing in a constraint are defined (required to test satisfaction). The following steps add the missing elements to an assignment. We call this the *completion* of the product selection. Assume that $f \in \text{FID}$, $a \in \text{AID}$, and feature model FM is encoded as constraints given by $\psi = \llbracket FM \rrbracket$.

1. Iterate the following steps until a fixed point is reached:
 - (a) If $f \in \text{dom}(\sigma)$ and f' is the parent of f , then set $\sigma(f') = 1$.
 - (b) If $f \in \text{dom}(\sigma)$ and f^\dagger appears in ψ , then set $\sigma(f^\dagger) = 1$.
2. If $f \notin \text{dom}(\sigma)$ and f appears in ψ , then set $\sigma(f) = 0$
3. If $f.a \notin \text{dom}(\sigma)$ and $f.a$ appears in ψ , then set $\sigma(f.a) = v$, where v is an arbitrary (integer) value within the range specified for $f.a$.

A product selection σ is *valid* whenever for all completions σ' we have $\sigma' \models \psi$.

Example 7. The product **P3** from Example 6 results in the following initial variable assignment

$$\sigma(\text{German}) = 1 \quad \sigma(\text{Repeat}) = 1 \quad \sigma(\text{Repeat.times}) = 10.$$

In the context of the feature model in Example 2. The remaining variables are `English`, `Dutch`, and `MultiLingualHelloWorld`, which is the parent of `Language` and `Repeat`, and there are no other attributes. The completion of σ includes the following additional elements:

$$\begin{array}{ll} \sigma(\text{MultiLingualHelloWorld}) = 1 & \sigma(\text{English}) = 0 \\ \sigma(\text{Language}) = 1 & \sigma(\text{Dutch}) = 0 \end{array}$$

The resulting completed assignment σ satisfies the constraints specified in Example 2. In contrast to this, the constraints would not be satisfied for product **P4**, where $\sigma(\text{English}) = 1$, $\sigma(\text{Repeat.times}) = 6$, and $\sigma(\text{Repeat}) = 1$, due to the clause $\text{English} \rightarrow (\text{Repeat} \rightarrow (\text{Repeat.times} \geq 2 \wedge \text{Repeat.times} \leq 5))$.

6 Product Generation

This paper introduced four language extensions to core ABS: the μTVL language to represent feature models, the delta modelling language (DML) to represent delta modules, the product line configuration language (CL) to associate deltas

to products and to establish the order of application of the deltas, and the product selection language (PSL) to describe the desired products. From a global perspective, these are used in the generation of a final software product as follows.

Given a core ABS module P , a set of delta modules Δ , a product line configuration C , a feature model FM , and a product selection p , the following steps are performed to build the final software product:

- Check** that the product selection p is satisfied by the feature model FM , as explained in Section 5.2.
- Select** the delta modules from Δ with valid application condition according to p , as described in Section 4.2.
- Apply** the deltas to the core module P , in the prescribed order, as described in Section 3.2. Add the initialisation block from the product selection—this will be the ‘main’ method.

Application of the deltas yields the final software product, a core ABS program.

7 Related Work

Existing approaches to express variability in modelling languages can be classified in two main directions [37]: annotative (or negative) and compositional (or positive). A third main approach for representing variability of development artefacts are model transformations.

Annotations. Annotative approaches consider one model representing all products of the product line. Variant annotations, e.g., using UML stereotypes in UML models [38,14] or presence conditions [12], define which parts of the model have to be removed to derive a concrete product model. The orthogonal variability model (OVM) proposed in Pohl. et al. [30] models the variability of product line artefacts in a separate model where links to the artefact model take the place of annotations. Similarly, decision maps in KobrA [1] define which parts of the product artefacts have to be modified for certain products. In the Koala component model [28], the variability of a component architecture containing all possible components is expressed by component parameterisation that is instantiated depending on the product features.

Composition. Compositional approaches, such as delta modelling [35,31,32,34], associate model fragments with product features that are composed for a particular feature configuration. A prominent example of this approach is AHEAD [3], which can be applied on the design as well as on the implementation level. In AHEAD, a product is built by stepwise refinement of a base module with a sequence of feature modules. Design-level models can also be constructed using aspect-oriented composition techniques [17,37,26]. Apel et al. [36] apply model superposition to compose model fragments.

Transformations. The common variability language (CVF) [16] represents the variability of a base model by rules describing how modelling elements of the base model have to be substituted in order to obtain a particular product model.

In [20], graph transformation rules capture artefact variability of a single kernel model comprising the commonalities of all systems. In [18], architectural variability is represented by change sets containing additions, removals or modifications of components and component connections that are applied to a base line architecture. Perrouin et al. [29] obtain a product model by model composition and subsequently refinement by model transformation.

Delta modelling. The notion of program deltas was introduced by Lopez-Herrejon [25] to describe the modifications of object-oriented programs. Schaefer et al. [35,31] introduced delta modelling as a means to develop product line artefacts suitable for automated product derivation. The conceptual ideas of delta modelling have also been applied the programming language level in an extension of Java with core and delta modules allowing the automatic generation of Java-based product implementations [32]. In recent work, Schaefer et al. [34,33] propose a version of delta-oriented programming where products are generated only from delta modules applied to the empty product. Furthermore, in this version the application conditions and the application ordering are specified separately from the delta modules in a product line specification in order to increase the reusability of the delta modules and to enable compositional type checking.

8 Conclusion

This paper presented the variability modelling fragment of the HATS ABS modelling framework, realised by languages μ TVL, DML, CL, and PSL. Together these languages can specify all the variability of a product line of core ABS models, with PSL scripts specifying the eventual products that can be derived.

The presented variability modelling concepts only target spatial variability. However, an ABS product line must also safely evolve over time in order to accommodate necessary changes after the deployment of the products; e.g., bug fixes, feature extensions or modifications, or changes in user requirements. In order to facilitate the modelling of temporal variability for core ABS models, it is crucial that evolution is expressed at the abstraction level of the modelling language. Hence, in the future, also within the scope of the HATS project, we are planning to extend the presented variability modelling concepts which are based on delta modelling with dynamic delta models to capture variability in space as well as variability in time.

A description of the core ABS language [15] and the proposed component model [24], along with a tutorial of the full ABS language and HATS tools suite [9] are available. In addition, the HATS tool suite, documentation, as well as several case studies are available from <http://www.hats-project.eu>.

References

1. Atkinson, C., Bayer, J., Muthig, D.: Component-Based Product Line Development: The KobrA Approach. In: SPLC (2000)
2. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# programming system: An overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 49–69. Springer, Heidelberg (2005)

3. Batory, D., Sarvela, J., Rauschmayer, A.: Scaling Step-Wise Refinement. *IEEE Trans. Software Eng.* 30(6) (2004)
4. Batory, D., Benavides, D., Ruiz-Cortés, A.: Automated analysis of feature models: challenges ahead. *Commun. ACM* 49(12), 45–47 (2006)
5. Benavides, D., Segura, S., Ruiz-Cortés, A.: Automated analysis of feature models 20 years later: a literature review. *Information Systems* (2010)
6. de Boer, F.S., Clarke, D., Johnsen, E.B.: A complete guide to the future. In: De Nicola, R. (ed.) *ESOP 2007*. LNCS, vol. 4421, pp. 316–330. Springer, Heidelberg (2007)
7. Boucher, Q., Classen, A., Faber, P., Heymans, P.: Introducing TVL, a text-based feature modelling language. In: *Proceedings of the Fourth International Workshop on Variability Modelling of Software-intensive Systems (VaMoS 2010)*, Linz, Austria, January 27–29, pp. 159–162. University of Duisburg-Essen (2010)
8. Burdy, L., Cheon, Y., Cok, D.R., Ernst, M.D., Kiniry, J.R., Leavens, G.T., Leino, K.R.M., Poll, E.: An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer, STTT* 7(3) (June 2004)
9. Clarke, D., Diakov, N., Hähnle, R., Johnsen, E.B., Schaefer, I., Schäfer, J., Schlatte, R., Wong, P.Y.H.: Modeling spatial and temporal variability with the HATS abstract behavioral modeling language. In: Bernardo, M., Issarny, V. (eds.) *SFM 2011*. LNCS, vol. 6659, pp. 417–457. Springer, Heidelberg (2011)
10. Clarke, D., Helvensteijn, M., Schaefer, I.: Abstract Delta Modeling. In: *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering, GPCE 2010*, pp. 13–22. ACM, New York (2010)
11. Classen, A., Boucher, Q., Heymans, P.: A text-based approach to feature modelling: Syntax and semantics of TVL. *Science of Computer Programming* (November 2010)
12. Czarnecki, K., Antkiewicz, M.: Mapping Features to Models: A Template Approach Based on Superimposed Variants. In: Glück, R., Lowry, M. (eds.) *GPCE 2005*. LNCS, vol. 3676, pp. 422–437. Springer, Heidelberg (2005)
13. van Deursen, A., Klint, P.: Domain-specific language design requires feature descriptions. *Journal of Computing and Information Technology* 10(1), 1–18 (2002)
14. Gomaa, H.: *Designing Software Product Lines with UML*. Addison Wesley, Reading (2004)
15. Hähnle, R., Johnsen, E.B., Schäfer, J., Schlatte, R., Steffen, M.: ABS: A core language for abstract behavioral specification. In: Aichernig, B.K., de Boer, F.S., Bonsange, M.M. (eds.) *FMCO 2010*. LNCS, vol. 6957, pp. 143–165. Springer, Heidelberg (2011)
16. Haugen, Ø., Møller-Pedersen, B., Oldevik, J., Olsen, G., Svendsen, A.: Adding Standardized Variability to Domain Specific Languages. In: *SPLC* (2008)
17. Heidenreich, F., Wende, C.: Bridging the Gap Between Features and Models. In: *Aspect-Oriented Product Line Engineering, AOPLE 2007* (2007)
18. Hendrickson, S.A., van der Hoek, A.: Modelling product line architectures through change sets and relationships. In: *ICSE*, pp.189–198 (2007)
19. Hirschfeld, R., Costanza, P., Nierstrasz, O.: Context-oriented Programming. *Journal of Object Technology* (March/April 2008)
20. Jayaraman, P.K., Whittle, J., Elkhodary, A.M., Gomaa, H.: Model composition in product lines and feature interaction detection using critical pair analysis. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) *MODELS 2007*. LNCS, vol. 4735, pp. 151–165. Springer, Heidelberg (2007)
21. Johnsen, E.B., Owe, O.: An asynchronous communication model for distributed concurrent objects. *Software and System Modeling* 6(1), 35–58 (2007)

22. Kang, K.C., Cohen, S., Hess, J., Nowak, W., Peterson, S.: Feature-Oriented domain analysis (FODA) feasibility study. Tech. Rep. CMU/SEI-90-TR-021, Carnegie Mellon University Software Engineering Institute (1990)
23. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J., Irwin, J.: Aspect-oriented programming. In: Aksit, M., Auletta, V. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
24. Lienhardt, M., Lanese, I., Bravetti, M., Sangiorgi, D., Zavattaro, G., Welsch, Y., Schäfer, J., Poetzsch-Heffter, A.: A component model for the ABS language. In: Aichernig, B.K., de Boer, F.S., Bonsange, M.M. (eds.) FMCO 2010. LNCS, vol. 6957, pp. 166–184. Springer, Heidelberg (2011)
25. Lopez-Herrejon, R.E., Batory, D.S., Cook, W.R.: Evaluating Support for Features in Advanced Modularization Technologies. In: Gao, X.-X. (ed.) ECOOP 2005. LNCS, vol. 3586, pp. 169–194. Springer, Heidelberg (2005)
26. Noda, N., Kishi, T.: Aspect-Oriented Modeling for Variability Management. In: SPLC (2008)
27. OMG: Unified modelling language, infrastructure and superstructure (version 2.2, OMG final adopted specification) (2009)
28. van Ommering, R.C.: Software reuse in product populations. *IEEE Trans. Software Eng.* 31(7), 537–550 (2005)
29. Perrouin, G., Klein, J., Guelfi, N., Jézéquel, J.M.: Reconciling Automation and Flexibility in Product Derivation. In: SPLC (2008)
30. Pohl, K., Böckle, G., Van Der Linden, F.: *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer, Heidelberg (2005)
31. Schaefer, I.: Variability Modelling for Model-Driven Development of Software Product Lines. In: Proc. of 4th Intl. Workshop on Variability Modelling of Software-intensive Systems, VaMoS 2010 (2010)
32. Schaefer, I., Bettini, L., Bono, V., Damiani, F., Tanzarella, N.: Delta-oriented programming of software product lines. In: Bosch, J., Lee, J. (eds.) SPLC 2010. LNCS, vol. 6287, pp. 77–91. Springer, Heidelberg (2010)
33. Schaefer, I., Bettini, L., Damiani, F.: Compositional Type-Checking for Delta-oriented Programming. In: Intl. Conference on Aspect-oriented Software Development, AOSD (to appear, 2011)
34. Schaefer, I., Damiani, F.: Pure Delta-oriented Programming. In: FOSD 2010 (2010)
35. Schaefer, I., Worret, A., Poetzsch-Heffter, A.: A Model-Based Framework for Automated Product Derivation. In: Proc. of Workshop in Model-based Approaches for Product Line Engineering, MAPLE 2009 (2009)
36. Apel, S., Janda, F., Trujillo, S., Kästner, C.: Model Superimposition in Software Product Lines. In: Paige, R.F. (ed.) ICMT 2009. LNCS, vol. 5563, pp. 4–19. Springer, Heidelberg (2009)
37. Völter, M., Groher, I.: Product Line Implementation using Aspect-Oriented and Model-Driven Software Development. In: SPLC, pp. 233–242 (2007)
38. Ziadi, T., Hélouët, L., Jézéquel, J.M.: Towards a UML Profile for Software Product Lines. In: van der Linden, F.J. (ed.) PFE 2003. LNCS, vol. 3014, pp. 129–139. Springer, Heidelberg (2004)

Automated Verification of Executable UML Models

Helle Hvid Hansen¹, Jeroen Ketema², Bas Luttik¹, MohammadReza Mousavi¹,
Jaco van de Pol², and Osmar Marchi dos Santos³

¹ Eindhoven University of Technology, Eindhoven, The Netherlands

² University of Twente, Enschede, The Netherlands

³ University of York, York, England

Abstract. We present a fully automated approach to verifying safety properties of Executable UML models (xUML). Our tool chain consists of a model transformation program which translates xUML models to the process algebra mCRL2, followed by symbolic model checking using LTSmin. If a safety violation is found, an error trace is visualised as a UML sequence diagram. As a novel feature, our approach allows safety properties to be specified as UML state machines.

1 Introduction

UML has become the popular modeling approach, driving the development of industrial applications in many different fields. One such field is the railway industry, where *Executable UML (xUML)* [35] is gaining popularity for specifying critical applications such as railway interlockings. The main goal of interlockings is to ensure that trains neither collide nor derail. This is achieved by establishing routes, which comprise tracks, points and other railway components, along which trains can pass safely. Correctness of interlockings is certainly imperative, and hence rigorous methods should be employed to verify their safety properties.

In this paper, we report on an automated approach to verifying safety properties of xUML models. These xUML models may specify the functional requirements of interlocking systems, but in principle they are not restricted to the interlocking domain. One of the target groups for our tool chain are modeling engineers with no training in formal methods. We accommodate this target group by allowing the functional requirements (i.e. the actual model) as well as their safety properties to be specified in xUML. The safety properties are expressed as state machines that “observe” the behavior of the model, and issue an error signal if a safety property is violated.

The verification is carried out using the well-known technique called model checking where the entire state space of a formal model is exhaustively explored and checked against a property. In our case, the formal model is specified in the process algebra mCRL2 [21] and the exploration is done using the symbolic model checking tools of LTSmin [8]. In particular, safety violations are found by detecting the above-mentioned error signals in the mCRL2 model. An xUML

model defines a generic model of object behaviour, but an mCRL2 model describes the behaviour of a concrete collection of interacting objects. Our tool chain therefore also takes as input an instance specification from which a model instance can be created. The verification is thus always carried out with respect to a particular model instance.

The mCRL2 model is obtained by automatically translating the xUML model, its safety properties and the instance specification into mCRL2. This automated translation is implemented using the Eclipse-based model transformation tools of Epsilon [30,31]. The translation goes via an internal format called iUML. This iUML representation is an intermediate step between the hierarchical xUML model and the “flat” transition system specified in mCRL2. We have several reasons for using such an intermediate representation. First, a one-step translation from xUML to mCRL2 would be quite complicated to implement due to the significant differences between the two languages. In particular, expressions and actions that can be statically evaluated are transformed in the iUML rather than translated into mCRL2 and evaluated there. Second, the iUML allows us to perform static analysis tasks which are not easily cast as a model checking task. Third, the iUML is sufficiently general to support different variations of the translation into mCRL2, and we expect that it can serve as a basis for translations into other target languages such as Promela [25] and Event-B [1].

The functionalities of our tool chain are illustrated in Figure 1 of the next section. To summarise, the tool chain consists of the following three main steps: (1) Automated translation of the model, its safety properties and an instance specification from xUML into the formal specification language mCRL2. (2) Checking for safety violations by searching for error signals in the mCRL2 model. (3) Visualisation of an error trace as an UML sequence diagram, in case a safety violation is found.

In the paper [22], we reported on the early developments of our approach. The main contributions of the present work with respect to [22] are:

1. A method for specifying safety properties as UML state machines.
2. A tool chain which realises a fully automated verification and feedback trajectory, where both input and output are expressed in UML.
3. Automation of the translation from xUML to mCRL2, and a more detailed description of the translation itself.
4. Investigations into the scalability of our approach by conducting verification of several different xUML interlocking models and more realistic track layouts (that yield model instances).

We believe that items 1 and 2 greatly improve the usability of our approach to modelling engineers and domain specialists, since only knowledge of UML is required, rather than familiarity with formal methods. We should point out that our current translation differs at some point from the translation in [22] (see end of Section 5).

The rest of this paper is structured as follows. In Section 2, we give a more detailed overview of our tool chain and its architecture. In Section 3, we introduce the subset of xUML supported by our tool chain, and discuss its syntax and

semantics. In Section 4, iUML is introduced, which is our intermediate format between the input models and the mCRL2 specification used for model checking. Section 5 presents the translation schema and Section 6 outlines the verification methods we used, the challenges we faced, and the results obtained. Finally, in Section 7 we conclude by discussing related and future work.

2 Tool Chain

The tool chain achieves its goal in a number of steps of which the details are hidden behind a rudimentary user interface. The core of this process is depicted in Figure 1. Next, we give more details on each part of the tool chain.

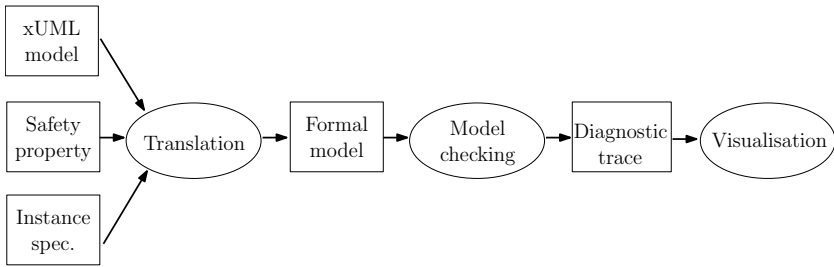


Fig. 1. Automatic verification of xUML models

Input. The inputs of our tool chain comprise two xUML models (specifying the model itself and its safety properties) and an instance specification (defining the object identifiers and their roles in associations). The family of xUML models that can be used as input are described in Section 3. The input xUML models must be provided in an XMI format that is compatible with the Eclipse UML Tools v3.0 (which implement the UML 2.2 specification). Currently we support two ways of generating these XMI files: An xUML model created in Artisan Studio can be transformed into XMI by our tool chain, or an xUML model can be created in the UML modeling tool Papyrus [37] which is based on Eclipse just as several other parts of our tool chain. Finally, the instance specification must be supplied in a simple text-based format that we defined for this purpose.

Translation. The automated translation from xUML to mCRL2 is implemented in the Eclipse Modeling Framework [41] (Eclipse Galileo version 3.5.2). In the Eclipse Modeling Framework (EMF), metamodels define the abstract syntax of EMF models. In close analogy, the UML superstructure [36] provides a meta-model that defines the UML language elements and their relationships. But unlike UML, EMF models generally do not have a graphical representation. As already mentioned, the automated translation goes via an intermediate representation called iUML, and we thus have metamodels that define the structure of

iUML models, and of their expressions and actions. We specify metamodels using the textual syntax of EMFText [23] which also provides a parser generator facility that we use to transform well-formed strings to EMF models. We access and manipulate EMF models using the Epsilon model transformation tools [30,31]: The transformation from UML to iUML is implemented in the model-to-model transformation language ETL. Generating mCRL2 code from iUML models is carried out using the model-to-text transformation language EGL.

Model checking. The generated mCRL2 specification is verified using a combination of the mCRL2 [21] and LTSmin [8] model checking tool sets (revision 8543 and the next branch dated 3-3-2011, respectively). From the mCRL2 tool set we use a number of utilities that pre-process the mCRL2 specification. In this pre-processing, parallel behavior is turned into non-deterministic sequential behavior and the result is simplified by removing redundant and constant data parameters. The latter step potentially reduces the size of the generated state space. The actual state space exploration is achieved using the symbolic reachability tool from the LTSmin tool set, which also provides distributed and multi-core reachability tools, and a tool to reduce state spaces modulo branching bisimulation.

Visualisation. In the case one of the specified safety properties is violated in the chosen instance of the xUML model, LTSmin generates a trace, that is, a sequence of actions leading to a violation of the property. This trace is visualized in Eclipse in the form of a UML message sequence diagram. If none of the safety properties are violated, a message is displayed reporting this result.

3 Executable UML: Translation Domain

In this section we describe the subset of Executable UML (xUML) [35] that is covered by our translation. For further information on the UML, we refer to [36].

3.1 Models, Classes and State Machines

Informally stated, an xUML model is a hierarchical structure that defines types of objects, their relationships and how they react to events in the system. In our subset of xUML, a *model* consists of signals, events, classes and associations. An event can be a *signal event* denoted simply by the signal name; a *change event*, denoted by `when(cond)` where `cond` is the change expression; or a (relative) time event, denoted by `after(n)` where `n` is the timeout delay. We do not include absolute time events. A *class* consists of properties, receptions and a state machine. A *property* can be an attribute, a generalisation or an association end. An attribute can be *derived* in which case it is defined by a Boolean expression. Derived attribute names start with a slash (/). A *generalisation* in a class `C` is a reference to another class. A class `C'` is called a superclass of `C` if `C'` can be reached from `C` via the transitive closure of the generalisation relation. We

require that attribute names are unique within a class and its superclasses. The *receptions* declare the signals that the class will react to, and the state machine specifies how the class reacts to events. State machines are described below. An *association* is an n -ary relation between classes.

Classes and their associations can be graphically represented by a class diagram, as illustrated in Figure 2 which shows the class diagram of a toy example called *Micro interlocking*, kindly provided to us by KnowGravity¹. Figure 2 shows that there are five classes: *track*, *point*, *signal*, *route* and HAL device where HAL device generalises *track*, *point* and *signal*. (The boxes with labels LCL and HAL can be ignored.) Furthermore, there are four binary associations, one between *route* and *track*, two between *route* and *point*, and one between *route* and *signal*. Note that a *route* instance should be linked to exactly one *signal* instance via the property *entry_signal*. Figure 2 also shows that in class *point* there are derived attributes called */at_left*, */at_right* and */is_locked*, but their definitions are not shown. Similarly, the classes *track* and *route* also have derived attributes.

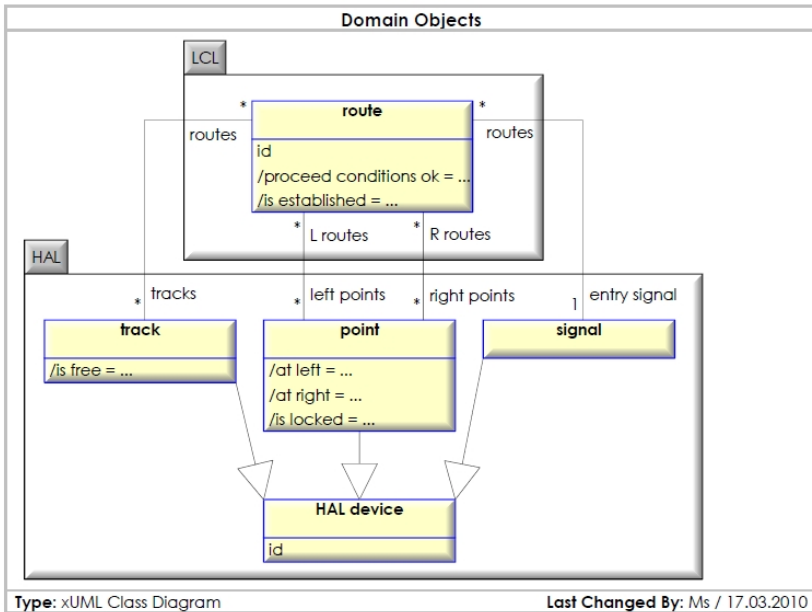


Fig. 2. Class diagram of Micro interlocking

A *state machine* consists of regions and states (alternatingly nested), and transitions between states. For a state s , we denote by $Regions(s)$ the regions immediately contained in s . Similarly, for a region r , we denote by $States(r)$ the states immediately contained in r . A *root region* is a region which is contained

¹ <http://www.knowgravity.com>

only in the state machine, but not in any state. A state may be *simple*, *composite* or *concurrent* meaning that it contains zero, at least one, or at least two regions, respectively. Furthermore, states may have *entry* and *exit* actions. The only pseudo-states we allow are *initial pseudo-states*, so in particular we exclude (deep) history, final, junction pseudo-states, and entry/exit points. The sets of all states and regions contained in the state machine of a class C are denoted by $States(C)$ and $Regions(C)$, respectively.

Example 1. The state machine diagram of `point` is found in Figure 3, and it shows, for example, that `working` and `moving` are composite, non-concurrent states (i.e. they contain one region), and the region of `working` contains the states `left`, `right` and `moving`. There are no concurrent states in `point`. All states except for `startup` and `moving` have entry actions. The tags, such as `<i>` or `<ic>`, that prefix signal names indicate a signal stereotype, but this is only a naming convention, and no UML semantics is derived from these tags².

A transition t goes from a source state $source(t)$ to a target state $target(t)$, and is labelled with $trg[grd]/eff$ where trg , grd , eff are the trigger, guard and effect of t , respectively. The trigger is an event, the guard is a Boolean expression, and the effect is a sequence of actions that should be carried out when the transition fires. We require that transition has a single event as trigger, but the guard and the effect can be omitted. Moreover, we require that for any state s there is at most one transition triggered by a time event with source s .

A transition t can be *internal* meaning that $source(t) = target(t)$ and firing t does not trigger exit or entry actions. In state machine diagrams, an internal transition in a state s is shown by placing the transition label $trg[grd]/eff$ in a box below the entry and exit actions of s . For example, in the `point` state machine there is an internal transition in the substate `left` of `working` with trigger `<ic> move right`.

If, for a transition t , $source(t)$ is an initial pseudo-state then $target(t)$ is called the *default entry state* of its enclosing region. If $source(t)$ is also directly contained in a root region, then t is called an *initial transition*. We require that initial transitions are unguarded.

A class inherits all properties, association ends, and so on, from its superclasses. In particular, a class also inherits state machines. So even though we only allow one state machine per class, when a class C is instantiated, the resulting object has all the state machines of C and its superclasses.

3.2 Expressions and Actions

We now describe the expression language and the action language for our models. These languages correspond to certain subsets of the SIML-language from [29].

Boolean expressions occur as transition guards, as change conditions, and as definitions of derived attributes. Apart from the usual Boolean connectives,

² The version of Artisan Studio used to create this model does not properly support UML stereotypes.

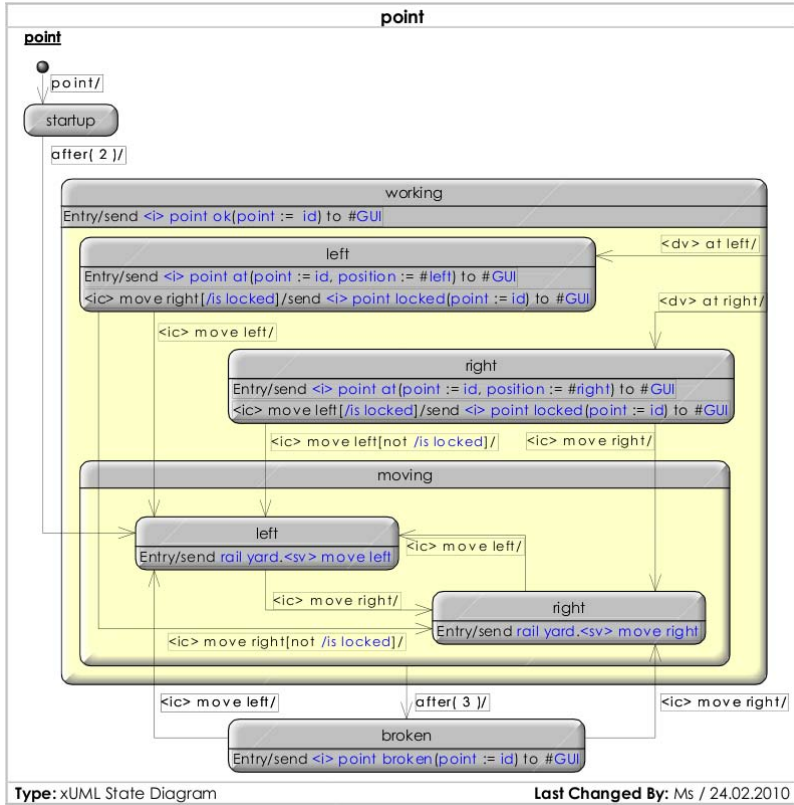


Fig. 3. State machine diagram of point in Micro interlocking

they may include quantification over linked objects thereby referring to non-local state. Formally, Boolean expressions are generated by the following grammar.

$$\begin{aligned}
 \textit{Bool} ::= & \textit{true} \mid \textit{false} \mid \textit{not Bool} \mid \textit{Bool and Bool} \mid \textit{Bool or Bool} \\
 & \mid \textit{in_state}(\#(\textit{State})) \mid \textit{DAttr} \\
 & \mid \textit{forall ObjExpr is_true Bool} \mid \textit{exists ObjExpr is_true Bool}
 \end{aligned}$$

$$\textit{ObjExpr} ::= \textit{AssocEnd} \mid \textit{AssocEnd UNION AssocEnd}$$

where *DAttr* and *AssocEnd* are identifiers of a derived attribute and an association end, respectively, and *State* is a sequence of dot-separated state or region names that describes a state of a given state machine. For example, the substate *left* of the state *moving* in the *point* state machine (see Figure 3) is referred to by the expression *working.moving.left*. Expressions of type *ObjExpr* denote sets of objects. Such sets can be specified either as the collections of objects that can be referenced via association ends, or as a union of such collections. The *forall* ... and *exists* ... denote quantification over sets of objects.

Examples of Boolean expressions in Micro interlocking are:

```

in point: in_state(#working.left)
in point: exists (L_routes UNION R_routes) is_true /established
in route: forall left_points is_true /at_left

```

There are two types of *basic actions*: assignments and sending signals. An object can send signals to itself, to linked objects, or to some environment interface (referred to as GUI) and they may carry a list of parameters. All actions can be composed sequentially. Formally, the action language is generated by the following grammar.

$$\begin{array}{ll}
 \text{Act} ::= \text{send } \text{Signal} & \text{Signal} ::= \text{SignalName} \\
 \quad | \text{send } \text{Signal} \text{ to } \#\text{GUI} & \quad | \text{SignalName}(\text{Params}) \\
 \quad | \text{send } \text{ObjExpr}.\text{Signal} & \\
 \quad | \text{Attr} := \text{Val} & \text{Params} ::= \text{ParamName} := \text{ParamVal} \\
 \quad | \text{DAttr} := \text{Expr} & \quad | \text{Params}, \text{Params} \\
 \quad | \text{Act}; \text{Act} &
 \end{array}$$

where *ObjExpr* is defined as above, *Attr* is the name of an attribute, *DAttr* is the name of a derived attribute, *ParamName* is an identifier and *ParamVal* is an expression that is evaluated at the time the action takes place. An action **send** *a* means that the object is sending the signal *a* to itself, and **send** *Obs.a* means that the signal *a* is sent to all objects in the set denoted by *Obs*.

3.3 UML Semantics

An xUML model defines a generic model of communicating objects. A model instance is obtained by instantiating classes and associations. For example, an instance of the Micro interlocking is defined based on a particular track layout which specifies a number of tracks, points, signals and routes and how they are linked (i.e. how the associations are instantiated).

UML semantics defines the operational behaviour of objects, that is, how objects react to events, and how they communicate with each other. Some of these aspects are defined by the UML specification [36], but for others the UML specification allows a choice between different interpretations in order to allow for flexible modeling. We give a brief, informal description of the semantics that we follow. We refer to [36, 15.3.12] for more details.

An *object of type C* is an instance of a class *C* together with an event pool. Due to inheritance, an object can have several state machines. These can be viewed as the concurrently executing regions of a single state machine. In the rest of this section, we let *O* denote an object of type *C*.

The set of states in which the object state machine currently resides is called the *active state configuration*. This is a set of states rather than a single state, due to the presence of concurrent regions. A transition is *enabled* if its trigger is available, its source state is active and its guard evaluates to true. When a

transition fires, its source state is exited (possibly triggering exit actions), then the effect actions are carried out, and finally its target state is entered, which again can trigger entry actions. It is possible that several transitions are enabled at the same time. In this case, a maximal set of consistent, enabled transitions is fired. Informally stated, two transitions are consistent if executing one does not disable the other by exiting its source state. This is, in particular, the case if the transitions are contained in disjoint regions of the state machine.

The UML specifies a *run-to-completion (RTC) semantics* which means that an object must finish all the behaviour triggered by an event, before the next event can be processed. While an object is processing an event its state is considered undefined, hence other objects are not allowed to inspect its state at this point. The behaviour imposed by the RTC semantics can be described by the following *object execution cycle*:

- O1: Let other objects read the currently active state configuration A , or choose an available event e for processing by the state machines of O . This marks the beginning of a run-to-completion (RTC) step in O .
- O2: Let T be a maximal, consistent set of enabled transitions in O with trigger e . If there are several such T s, then one is chosen nondeterministically. Fire all transitions in (the possibly empty) T (in arbitrary order). The RTC step is now completed. Go to step O1.

Note that in agreement with UML 2.2 semantics (see *Run-to-completion and concurrency* in [36, 15.3.12]) the RTC step applies simultaneously to all regions of the state machine. But the RTC steps of different objects may be interleaved.

We now define when events are available for processing: A *signal event* denotes the moment when a signal is sent. A signal event is available if it is in the event pool. A *change event when(cond)* is available whenever the change condition cond is true. This seems to conflict with [36, Sec.13.3.7] which says that a change event occurs when the change condition *becomes* true. But UML 2.2 does not specify when change events are detected, or whether they remain after the change condition becomes false again. In our interpretation change events are thus detected immediately, but they do not remain. A *time event after(n)*, which triggers a transition t , is available whenever $\text{source}(t)$ is active. This semantics is based on the assumption that all transitions and actions take place in zero time.

The UML specification allows for different priority schemes when choosing an event for processing by an object, see [36, Sec.15.3.12]. We apply the following priority scheme: signals from the object to itself have priority over all others; signals from the environment, time events and change events have priority over signals coming from other objects; signals coming from other objects are processed on a FIFO basis (which we realise by implementing the event pool as a queue). Note that the semantics does not impose any fairness constraints: an available event is not guaranteed to be processed. In terms of transitions, it means that a transition may be enabled but never taken.

Objects communicate by sending signals to each other. We assume that signals are never lost or duplicated. The communication is one-to-one (i.e. no broadcast) and asynchronous (since signal events are stored in event pools).

4 The iUML Representation

The iUML representation can be described as an intermediate step between UML and a labelled transition system (LTS) representation of object behaviour. The LTS states are active state configurations and labelled transitions are defined according to the semantics described in Section 3.3. Furthermore, we use the iUML to represent model instances.

4.1 Transitions in iUML

In order to associate a unique action sequence with iUML transitions it may be necessary to refine UML transitions into several iUML transitions. The reason is that a UML transition t can have a composite source state, hence the (exit) actions that should be executed when firing t may depend on which nested substates of $source(t)$ are active. We illustrate the refinement of such a transition with the following simple example. Consider the transition t from A0 to B0 in Figure 4. (We have only drawn the elements relevant for the refinement of this t .) Assume that A0 is active, the trigger of t is available, and its guard is true.

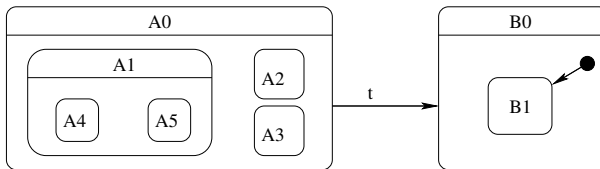


Fig. 4. UML transition with composite source state

In order to determine which actions to carry out when firing t , we need to also inspect which of the states A1, A2, A3, A4 and A5 are active and whether they have exit actions defined. Suppose that A0, A4 have exit actions a_0, a_4 , respectively. We then have two possible exit action sequences when t fires: if A4 is active, the exit action sequence is $a_4; a_0$ and if A4 is not active, it is a_0 . This motivates the following definitions.

- An *active state predicate* is a pair (U, V) where $U, V \subseteq States(C)$. An active state configuration A satisfies (U, V) if $U \subseteq A$ and $V \cap A = \emptyset$. In other words, A satisfies (U, V) if all states in U are active, and no states in V are active. We denote by $\llbracket (U, V) \rrbracket$ the set of active state configurations of O that satisfies (U, V) .

- An *iUML transition* consists of an active state predicate (instead of a source state), a target state, a trigger, a guard, an exit sequence, an effect sequence and an entry sequence. An *iUML transition* t is *enabled* if the currently active state configuration satisfies the active state predicate of t , the trigger is available and the guard is true.

Due to the nesting of A4 inside the composite state A1, the transition t from Figure 4 is refined into three *iUML transitions* t_1, t_2 and t_3 with the following active state predicates and exit sequences:

flat	active state predicate	exit sequence
t_1	$(\{A0, A1, A4\}, \emptyset)$	$a_4; a_0$
t_2	$(\{A0, A1\}, \{A4\})$	a_0
t_3	$(\{A0\}, \{A1\})$	a_0

Note that in *iUML transitions*, we keep as much of the high-level representation of the UML as we can. For example, in the above we have one *iUML transition* t_3 rather than two (where one requires A2 to be active, and the other one requires A3 to be active). This is facilitated by the use of active state predicates instead of source states. Note that the entry sequence of a UML transition does not depend on the currently active state configuration. For example, if in Figure 4 B0, B1 have entry actions b_0, b_1 , respectively, then the entry sequence is $b_0; b_1$ for all t_1, t_2, t_3 .

4.2 Transition Selection

Recall from the object execution cycle (Section 3.3) that upon receiving an event e , an object must select a maximal, consistent set of enabled transitions for execution. In the *iUML*, transitions are grouped together in a way that reflects the transition selection algorithm. We first group an object's transitions by trigger and active state predicate. For each such transition group T , the consistent subsets of T that can fire depend also on the values of transition guards. These subsets are called multi-transitions. In the *iUML* representation, each object will contain a collection of transition groups, and each transition group will contain its multi-transitions.

We now give a formal definition. Let O be an object of type C . For an event e , we let $Tr(O, e)$ be the set of all transitions in O with trigger e . A *transition group with active state predicate* (U, V) and trigger e (in object O) is a non-empty, maximal subset T of $Tr(O, e)$ such that for all $t \in T$, $\llbracket(U, V)\rrbracket \subseteq \llbracket(U_t, V_t)\rrbracket$ where (U_t, V_t) is the active state predicate of t . Note that if e is a time event or a change event, then $Tr(O, e)$ is a singleton and equal to the only transition group with trigger e . Transition groups that contain several consistent transitions can result from transitions in concurrent regions.

For example, an object of type `point` has five transitions triggered by the signal `<ic> move left` (see Figure 3). These transitions will result in four transition groups. One group will contain the two transitions that both have source

state `working.right`; the other three groups will be singletons. Note that the non-singleton transition group is not consistent (since firing one will disable the other by exiting `working.right`). However, these transitions will never be enabled at the same time, since their guards are complementary.

The subsets of a transition group that can actually fire depends on the consistency of transitions and the value of transition guards. We formalise these subsets as multi-transitions. A *multi-transition* of a transition group T is a consistent subset M of T such that $M = G \cup \text{ungrd}(T)$ where $G \subseteq \text{grd}(T)$, $\text{grd}(T)$ is the set of all guarded $t \in T$, and $\text{ungrd}(T) := T \setminus \text{grd}(T)$.

For example, if $T = \{t_0, t_1, t_2\}$ is a transition group in which all transitions are pairwise consistent, and $\text{grd}(T) = \{t_1, t_2\}$ where g_i is the guard of t_i for $i = 1, 2$, then the multi-transitions of T are $\{t_0, t_1, t_2\}$, $\{t_0, t_1\}$, $\{t_0, t_2\}$ and $\{t_0\}$ which correspond to the scenarios where g_1, g_2 are both true, only g_1 is true, only g_2 is true, or g_1, g_2 were both false.

5 Translation from iUML to mCRL2

We sketch our translation from iUML into mCRL2. The mCRL2 specification language [21] extends the process algebra ACP [5] with facilities to specify abstract datatypes (ADTs), and includes built-in types such as Booleans, integers, and lists. The advantages of translating into mCRL2 are that the language has a formal semantics [21], and that it comes with powerful verification technology. The possibility of defining data types and using them in the behavioral specification is essential for the translation presented in this section. Process algebras without ADT support such as CSP [24] supported by the FDR2 toolset [15] are thus not directly applicable, while process algebras with ADT support such as E-LOTOS [26] and LOTOS-NT [40] supported by the CADP toolset [16] provide alternatives for mCRL2 in our setting. We refer to [21] and the website <http://www.mcrl2.org> for details regarding the syntax and operational semantics of mCRL2.

Our translation takes as input an iUML model instance \mathcal{M} . \mathcal{M} describes a collection of objects that interact by sending signals to each other, but also by inspecting each others state. We translate each object into an mCRL2 process; the translation of the entire model \mathcal{M} will then, roughly, be the parallel composition of the mCRL2 processes associated with the objects in \mathcal{M} , and the interaction between the objects is implemented by means of mCRL2's facility for (synchronous) communication between components.

We proceed to discuss the translation of the objects in \mathcal{M} as *object processes*. Recall that each object has associated with it an event pool, and a collection of state machines describing its behaviour. An object process will consist of the parallel composition of a buffer process modelling the event pool, and a state machine process modelling the actual behaviour of the object. Below we will discuss the translation of the event pool and object behaviour in more detail, illustrating the translation on the object `p1` of type `point` (see Fig. 3).

Event Pool. Signals are specified using mCRL2's facility for defining abstract data types. The signals that are to be stored in the event pool of an object of type C are represented by an enumerated data type $C_Messages$. That is, $C_Messages$ has a member for each reception in the class C . The buffer process of an object of type C receives and stores the signals that are sent to the object by implementing a queue of $C_Messages$.

Example 2. The specification of the enumerated type `point_Messages` is:

```
point_Messages = struct
  ic_move_right_point | ic_move_left_point |
  dv_at_right_point | dv_at_left_point ;
```

The event pool of the object `p1` is specified by the following mCRL2 process specification.

```
proc point_Buffer_p1(message_buffer: List(point_Messages))=
  % Messages received from other components
  (#message_buffer < max_buffer_size) ->
    sum m: point_Messages. sum sender : Identifiers.
      receive_from_system(sender, p1, m).
      point_Buffer_p1(message_buffer <| m)
+ % Messages sent to component
  (#message_buffer > 0) ->
    send_to_component(p1, head(message_buffer)).
    point_Buffer_p1(tail(message_buffer));
```

The specification above expresses that the process `point_Buffer_p1` consists of a choice (denoted by `+`) between the following options: either (if its buffer is not full) it receives some element `m` of sort `point_Messages` from some `sender` and then appends `m` to its `message_buffer`, or (if its buffer is not empty) it sends the first element in `message_buffer` to the associated state machine, removing this element from the `message_buffer`.

States, Messages and Buffers. The state machine process implements the concurrent composition of the state machines of the class C , and the process carries data parameters that encode the active state configuration by letting each such state parameter range over an enumerated data type that represents the states contained in a region r . More precisely, we declare for each state machine X in C and each region r in X , an enumerated data type X_r_States whose members represent the set $States(r)$. If r is not a root region, then X_r_States will also have a member (ending with `_nop`) that will be used to indicate that no state in r is currently active.

Example 3. The regions in the Point state machine shown in Figure 3 give rise to enumerated data types: `point_States`, `point__working_States`, and `point__working_moving_States`. Below we show the definitions of the first two:

```

point_States = struct
    point__broken_substate
| point__startup_substate
| point__working_substate ;

point__working_States = struct
    point__working_right_substate
| point__working_left_substate
| point__working_moving_substate
| point__working_nop ;

```

Recall from Figure 2 that Point is a specialisation of HAL device; the active state configurations of the latter are represented by the data type `HAL_device_States`. The behaviour defined by the state machines of the object `p1` is then expressed by a process definition of the form

```

proc point_p1(
    HAL_device_state : HAL_device_States,
    point_state : point_States,
    point__working_state : point__working_States,
    point__working_moving_state : point__working_moving_States
) = ...

```

We postpone the discussion of how the state machines of HAL device and Point give rise to the right-hand side of the above defining equation for `point_p1`. First, we explain how mCRL2's features of parallel composition, communication and blocking are used to combine `point_Buffer_p1` and `point_p1` specifying the behaviour of the object `p1`. The object `p1` is represented by the following expression:

```

proc point_Complex_p1
= block({send_to_component, receive_from_buffer},
    comm({send_to_component|receive_from_buffer -> message_to_component},
        point_Buffer_p1([])
    || point_p1(HAL_device__normal_substate, point__startup_substate,
        point__working_nop, point__working_moving_nop)));

```

The process `point_Complex_p1` is defined as a parallel composition of instances of the processes `point_Buffer_p1` and `point_p1`. Initially, the active states are `startup` (in Point) and `normal` (in HAL device), and the buffer is empty; this explains the respective parameter values passed to `point_Buffer_p1` and `point_p1`. The operation `comm` expresses a communication between `point_p1` and `point_Buffer_p1`: both components may synchronise by simultaneously executing the action `receive_from_buffer` and `send_to_component`; the resulting event is denoted by `message_to_component`. The operation `block` declares, in fact, that the actions `send_to_component` and `receive_from_buffer` may not be executed in isolation; they may only occur as part of the aforementioned synchronisation.

State Machine Process. We proceed to explain how the behaviour of an object as expressed by a state machine is translated into mCRL2. The state machine process implements the object execution cycle (see page 233). Starting from a “stable state”, which corresponds to *O1* in the object execution cycle, the process can either receive and process an event, or let other processes inspect its state. This inspection of states is implemented as a communication of state parameter

values: a consumer action takes place in the object process that needs the data; the matching producer action takes place in the object process whose current state must be known to the consumer.

The possible behaviours in state *O1* (process event or send state data) are modelled as a nondeterministic choice (sum) over the different alternatives. As an example, part of the state machine process for the object *p1* is sketched below. In the first two summands, a message is received from the buffer. In the next two summands, a message is received directly from the environment. The following two summands represent the two time event transitions in the Point state machine, one has source state *startup* and the other has source state *working.moving*. The last four summands are actions that produce data for the evaluation of change conditions in two Route objects *r1* and *r2*. They show that in *r1*, there are two change conditions that require the expression *in_state(#(working.right))* to be true in *p1*.

```

proc point_p1(<state params>)
  = receive_from_buffer(p1,ic_move_left_point). ...
  + receive_from_buffer(p1,ic_move_right_point). ...
  + receive(p1,dv_at_left_point). ...
  + receive(p1,dv_at_right_point). ...
  + (point_state == point__startup_substate) ->
    tick(p1). ...
  + (point__working_state == point__working_moving_substate) ->
    tick(p1). ...
  + when_data_r1_1_p1_producer(p1,
    point__working_state == point__working_right_substate). ...
  + when_data_r1_2_p1_producer(p1,
    point__working_state == point__working_right_substate). ...
  + when_data_r2_1_p1_producer(p1,
    point__working_state == point__working_left_substate). ...
  + when_data_r2_2_p1_producer(p1,
    point__working_state == point__working_left_substate). ...

```

Recall that in our semantics, time event transitions can fire whenever the source state is active. The detection of a time event is specified by the action *tick*. The *tick* action does not actually place an event in the buffer. Change events are specified in a manner similar to time events which we describe towards the end of this section.

The part of the state machine process that follows an action that models the choice of an event for processing specifies the transition selection algorithm and the execution of the selected transitions. Recall now that in the iUML representation, we have a representation of the transition selection algorithm given by transition groups and multi-transitions. The mCRL2 specification that implements the transition selection algorithm consists of a nesting of conditional statements ranging over transition groups. For each transition group it is checked whether its active state predicate is satisfied by the currently active state configuration. If this check fails for all transition groups, then the process continues recursively with its state parameters unchanged. Otherwise, there is a transition

group T for which the active state predicate is satisfied by the currently active state configuration. If the guards of transitions in T refer to the state of other objects, then the process carries out a number of consumer actions to retrieve this state information. Next, a conditional statement runs through the multi-transitions of T ordered decreasingly by size, until it finds a multi-transition M whose transition guards are all true, and then the action sequence associated with M is executed, and the state machine process continues recursively with its state parameters updated to reflect the state after firing the transitions in M .

The mCRL2 code for the transition group in p_1 with trigger `move_left` and active state predicate ($\{\text{working.right}\}, \{\text{working.left, working.moving.right, broken}\}$) is shown in Figure 5. The transition group consists of two transitions t_{left} and t_{right} with guards `/is_locked` and `/is_locked`, respectively. Such pairs of transitions with complementary guards result in two singleton multi-transitions. The derived attribute `/is_locked` in p_1 refers to the state of route objects r_1 and r_2 which explains the communication with r_1 and r_2 in lines 2-4. These consumer actions are matched by producer actions in r_1 and r_2 . In line 3, the active state predicate is checked. In line 7 the transition guard `not /is_locked` is evaluated using the received data values. Line 8 contains the action that results from firing the multi-transition $\{t_{\text{left}}\}$. Lines 9-13 specify the updated state of the process after firing $\{t_{\text{left}}\}$. If `not /is_locked` evaluates to false in line 7, then `/is_locked` must evaluate to true, and so the multi-transition $\{t_{\text{right}}\}$ is fired, which is specified in lines 14-20.

```

1 point_p1(...) =
2   ...
3   (point__working_state == point__working_right_substate) ->
4   (sum r1_var: Bool. sum r2_var: Bool.
5     condition_data_p1_1_consumer(r1,r1_var)
6     | condition_data_p1_1_consumer(r2,r2_var).
7   (!(r2_var || r1_var)) ->
8     send_to_rail_yard(p1,sv_move_left_point_railyard).
9   point_p1(
10     HAL_device_state,
11     point_state,
12     point__working_moving_substate,
13     point__working_moving_left_substate)
14   <>
15   send_to_environment(p1,i_point_locked_point_environment).
16   point_p1(
17     HAL_device_state,
18     point_state,
19     point__working_state,
20     point__working_moving_state)
21   )

```

Fig. 5. Example mCRL2 code for a transition group

A transition t_c triggered by a change event c can fire whenever the change condition is true, and $source(t_c)$ is active. If the change condition refers to data in other objects, then a sequence of consumer actions are executed in order to obtain the data, similarly to how transition guards are evaluated.

Difference with Earlier Translation. In our earlier work [22], we presented a slightly different translation from xUML to mCRL2. This translation was not formulated in terms of an intermediate format, as it was done by hand. There are, however, also semantic differences between the two translations:

- In the translation from [22], change events are detected by an additional monitor component of object processes, and when a change condition becomes true, the monitor places a message in the buffer. In particular, change events remain in the buffer even after the condition becomes false.
- In the translation from [22], change events, object-internal signals, and system-internal signals have equal priority (all go through the FIFO-buffer), and all these events take priority over external signals. In the current translation, object-internal signals have priority over all others, and change events, time events and external events are allowed to overtake system-internal signals.

Based on discussions with the UML modelling engineers, our current translation is more in line with their view on UML semantics (which is based on the CASSANDRA simulator [29]) than our previous translation. As a further advantage, we have found that the mCRL2 models resulting from our current translation are dealt with more easily by our model checking tools.

6 Verification

Our approach to verifying safety properties of xUML models is based on expressing the safety properties as UML state machines that observe the system state and send an error signal in case a violation is found. The model and its safety properties are both translated into mCRL2, as described in the previous section, and safety violations are detected by using the facility of our model checking tools that allow searching for a particular action, in this case, the “*send error signal*”-action.

Our motivation for expressing safety properties as state machines is two-fold. First, it allows UML modelling engineers to specify safety requirements without having to learn temporal logic. Second, the mCRL2 tools that provide (explicit-state) model checking of modal mu-calculus formulas were not able to deal with the sizes of our models. By turning the verification problem into a reachability problem, we were able to verify our models using the symbolic reachability tool from the LTSmin tool set [8]. This symbolic tool allows for varying exploration strategies, and reports some basic performance analytics.

In Section 6.1 we describe how safety properties are modelled as UML state machines. The verification proper is discussed in Sections 6.2 and 6.3, which investigate, respectively, the size of the models we are able to deal with, and ways to ‘attack’ larger models.

6.1 Safety Properties as Observer Classes

In an internal document describing the Micro interlocking the following two safety requirements are given:

MS1: “A point that is locked by an established route shall never move.”

MS2: “The entry signal of a route shall never display proceed when one of its tracks is not free.”

The exact meaning of the railway signaling concepts mentioned in MS1 and MS2 is not so important for the present discussion, but one should think of a route being established as a requirement for letting a train pass (safely) over the route. What is relevant is that properties such as a point being locked, a route being established, and hence MS1 and MS2 themselves, can be expressed in terms of the system’s state, that is, without reference to the ordering of events. We will use the term *state property* to refer to such safety properties.

Our approach to verifying state properties is based on the observation that state property violations can be detected in the system itself as certain change events. In order to detect such violations we define a collection of *observer classes* whose state machines will detect safety violations and send error signals. These observer classes are expressed in the same subset of xUML as the model we wish to verify, and we can therefore apply our automated translation to generate an mCRL2 specification of the “observed model”.

Observer Classes. The common structure of observer classes is modelled by the class `StateObserver`. The `StateObserver` class has three attributes: `id`, `ObservedObject` and `ObservedClass`, one derived attribute `/triggered` (which will be defined by a Boolean expression), and a state machine with a single transition with trigger `when(/triggered)` and effect `send <i>_violation(observer := id) to #GUI`. The purpose of the attribute `ObservedClass` is to define the context in which `/triggered` is evaluated. The attribute `ObservedObject` must be the name of an instance of `ObservedClass`. All attribute values are defined upon instantiation.

Specific state properties are modelled as specialisations of the class `StateObserver`. These specialisations are what we call *observer classes*. The state machine of an observer class consists of just one state and an initial transition to it. The definition of `/triggered`, and of any additional derived attributes that may aid the definition of `/triggered`, are assigned as the effect of the initial transition. We illustrate using the two examples MS1 and MS2 from above. It should be clear that MS1 and MS2 are both state properties.

The property MS1 will be modelled as an observer for the class `Point`, that is, the attribute `ObservedClass` has value “`Point`”. Hence, the definition of `/triggered` may use derived attributes from the `Point` class such as `/is_locked` which is defined as `/is_locked := exists (L_routes UNION R_routes) is_true /is_established`. In other words, `/is_locked` is true if the point is locked by an established route, and we define `/triggered := /is_locked and in_state(#working.moving)`.

The property MS2 is modelled as an observer for the class Route, and it defines two “auxiliary” derived attributes `/all_tracks_free` and `/proceed`. The initial transition of its state machine has effect:

```

/all_tracks_free := forall tracks is_true /is_free;
/proceed := entry_signal.in_state(#proceed);
/triggered := /is_established and /proceed and not /all_tracks_free;
    
```

6.2 Feasibility of Verification

Given instances of the translated UML models, we would first of all like to know the size of models we can deal with. To this end we designed several track layouts for the Micro interlocking which are of increasing complexity. The layouts are presented in Figure 6 and correspond to simple configurations one might find in rail yards. Although not depicted in the figure, the possible routes in a layout are precisely all the maximal paths in starting from a signal and not passing both the left and right branch of a point (e.g., Layout 5 has six routes).

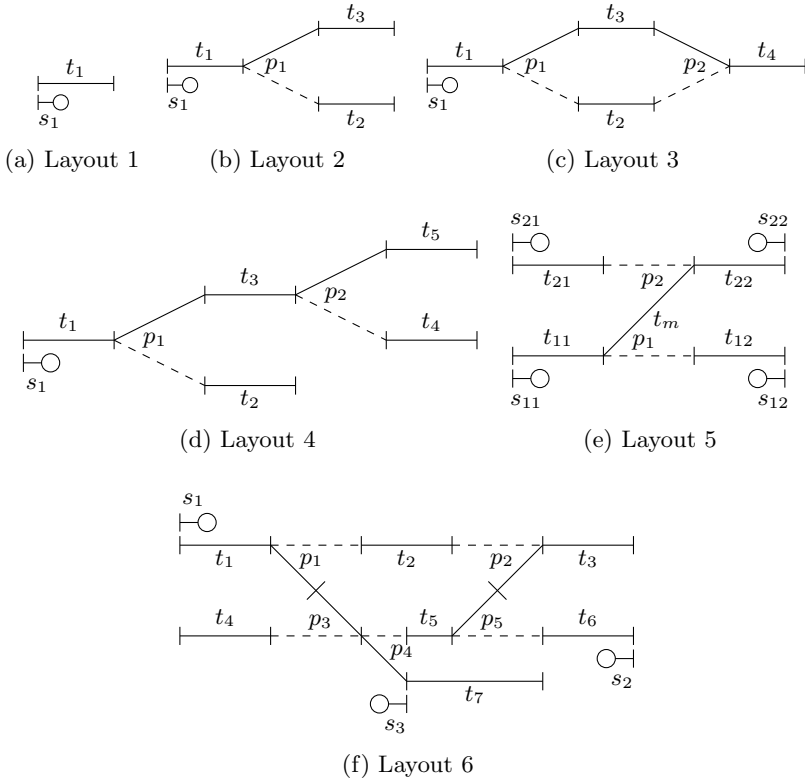


Fig. 6. Several track layouts used to test the feasibility of the verification task

Using the mCRL2 and LTSmin model checking tool sets (see Section 2) we can next generate the state spaces for the depicted track layouts. The measurements as obtained with the BDD based symbolic model checker from the LTSmin tool set are depicted in Table 1. As can be seen in the table, model checking our simple track layouts is possible, but running times and memory consumption increase fast when introducing more routes (compare Layouts 4 and 5, where 4 has three routes and 5 has six).

Table 1. State spaces of the layouts from Figure 6, without observers, where each state machine is assumed to have at most one message in its event pool. Resource consumption is for the LTSmin symbolic model checker run on an Intel Xeon X5550 machine with 148 GB of internal memory; a saturation-like [9] exploration strategy was used. The running times exclude the time to load the model.

Layout	Components	Routes	States	Runtime (s)	Memory (MB)
1	2	1	1.7×10^4	0.01	61
2	5	2	1.3×10^9	0.25	76
3	7	2	4.9×10^{11}	7.73	86
4	8	3	8.9×10^{13}	19.39	115
5	11	6	6.8×10^{23}	2605.90	3133
6	15	8	$> 7.0 \times 10^{30}$	> 496 h	> 30 GB

Given the resources consumed already by Layout 5, we can conclude that it is impossible to generate complete states spaces for ‘realistic’ layouts, which usually consists of hundreds of components and routes. Hence, more advanced methods are required, which are discussed in the next section. Of course, if certain properties are violated, this might already be detectable when instantiating the UML models for the small layouts.

Remark 1. The figures in Table 1 make it clear that explicit state space generation is infeasible. Using the distributed model checker which is also part of the LTSmin tool set, already Layout 2 is too large to be dealt with on a cluster of 10 Intel Xeon E5335 machines each with 24 GB of internal memory.

Given the simplicity of track layouts and the state machines shown, it may come as a surprise that the state spaces are so huge. Note, however, that not all states are depicted in the state machines: First, event pools are not included. Second, as a sequence of actions that is carried out when a transition fires is not executed as an atomic block, additional ‘intermediate’ states exist between actions in the sequence. Third, the communication needed to exchange state parameters also introduces some additional states.

Remark 2. Bounded model checking (BMC) [6] is not a suitable model checking technique in the current context. The technique cannot prove the absence of errors, which is precisely what we are interested in, given the safety critical nature of interlockings.

6.3 Speeding and Scaling Up Verification

Given our interest in proving the presence or absence of certain actions (see Section 6.1), it might be possible to speed up and scale up the verification. From the literature at least two symbolic model checking techniques are known that may help to achieve this:

- Compositional exploration of state spaces [33,4]: The transition relation is split into several parts and only some of these are used in state space exploration. Selection of the employed parts is based on an analysis of the interaction between the parts and on the particular property one is interested in. Additional parts are selected in case the property could be neither proved nor disproved using the selected subset of the transition relation.
- Counterexample-guided abstraction refinement (CEGAR) [11]: Given a property, the transition relation is over-approximated (i.e., a relation is used of which the transition relation is a sub-relation). Next, refinement takes place based on violations of the property. Eventually either a violation is found that also holds given the transition relation (i.e., the non-over-approximated one) or an over-approximation is reached in which the property holds (implying it also holds given the original transition relation).

In [33,4,11] it is reported that with both techniques, speed up and scalability are achieved in case only part of the transition relation is needed to show that the considered property holds. Moreover, in case of CEGAR speed up and scalability are also achieved as the symbolic representation of the over-approximated state space is often smaller than the symbolic representation of the real state space.

Preliminary Results. Thus far we have extended the symbolic model checker from the LTSmin tool set with the first of the aforementioned techniques and we are currently working on implementing the second technique.

Some preliminary results obtained with the implementation of the first technique mentioned above are shown in Table 2. The safety properties MS1 and MS2 (see Section 6.1) have been verified for Layout 2 from Figure 6. Both properties are violated by the Micro interlocking specification and, hence, an error trace (of a certain length) can be generated.

Table 2. Running times and trace lengths when searching for error actions using the LTSmin symbolic model checker run on an Intel Core 2 Duo machine with 4 GB of internal memory; a saturation-like [9] exploration strategy was used. The running times include the time to load the model and generate a trace.

Layout	Property	Runtime (s)		Trace Length	
		Default	Compositional	Default	Compositional
2	MS1	6.33	6.42	35	22
	MS2	7.33	6.78	41	41

It is impossible to draw any definite conclusions given the small sample. Nevertheless, we note that some speed up is obtained in the case of MS2. Furthermore, the reported lengths of the error traces are encouraging: the length of the error trace is substantially shorter in the case of MS1, and it is not longer in the case of MS2.

Remark 3. We do not automatically obtain the shortest trace possible, as we use a saturation-like strategy instead of breadth-first search. Using breadth-first increases running times for the layout 2 to 7.05 and 8.98 seconds for MS1 and MS2, respectively. For layout 2, the shortest error traces for MS1 and MS2 consist, respectively, of 22 and 28 steps.

7 Discussion and Conclusion

We have presented a fully automated, translation-based approach to the verification of safety properties in xUML models. Since both the input and the output of our tool chain are expressed in UML, our verification technology can be used by engineers without a thorough background in process algebra, model checking, or modal mu-calculus.

Additional Case Studies. Our translation from xUML to mCRL2 has been further applied in two case studies: (i) a UML model of a controller for mixing hot and cold water, obtained from one of the industrial partners of the University of Twente; (ii) a UML model of the session setup protocol from the ISO/IEEE 11073-20601 [27] standard (a data exchange standard for the health care industry).

With regard to item (i), we were able to identify certain property violations. However, this required the use of temporal logic, as the specified properties were liveness properties (which we currently cannot capture using our observer state machine approach). Moreover, since the controller for mixing hot and cold water is intended to be implemented on a Programmable Logic Controller (PLC), which does not buffer incoming events, we modified our translation in this respect.

The second case study (ii) was carried out by a colleague from the Eindhoven University of Technology [28]. After resolving some ambiguity issues in the state machines provided, the UML model could be translated into mCRL2. The verification revealed that it is possible for the system setup to reach an unsafe state (where the communicating devices operate with different measurement units). However, it was also shown that no unsafe operational behaviour could occur, due to detection of the unsafe state before the first data exchange.

Related Work. Formalisation of xUML models for the purpose of verification is a widely studied topic, and we mention just a few [2,3,12,20,34,42,44]. We briefly relate our approach to some of the aforementioned ones. Recall that we currently support signal, time and change events, but not call events.

In the UMC framework of [3], xUML models may be specified in the UMC specification language. UMC supports signal and call events, but no time or

change events, nor exit/entry actions of states. Properties must be specified in a CTL-like logic, and are verified using on-the-fly model checking. Specifying properties of UMC models thus requires some knowledge of formal methods (and not just of UML), but on the other hand, the types of properties that can be expressed far exceeds what our observers can define. In [20], xUML models (with change, call and send events) are translated into a temporal logic that supports compositional specification, and verification takes the form of refinement proofs. Our approach is based on model checking which has the advantage that the verification process is fully automatic whereas theorem proving generally requires human interaction in order to find proofs. In [34], UML state machines are translated into timed automata using, as we do, model transformation technology. The resulting timed automaton is verified using the UPPAAL model checker. It is, however, not clear how the timing constraints of the timed automaton are derived from the input xUML model, in particular, since only signal events are allowed as transition triggers (see Rule 10 in [34]).

Formal verification of high-level interlocking specifications has been studied in [13,43]. For verification of concrete interlocking systems, see e.g. [10,14,19]. One major source of challenges in the verification of xUML models lies in the asynchronous communication model. It has been observed earlier that synchronous specifications may be more amenable to automatic and exhaustive verification; see, e.g., [39], for a short experience report in the domain of railway interlockings.

Our approach to modelling safety properties of xUML models in xUML itself seems to be new, but similar ideas are found in [7,17,18,32,38]. The main difference with our work is that our observers monitor state changes whereas the above-mentioned observers monitor event sequences, and hence they are able to express temporal properties. The expression of liveness properties in our approach is still very much an open issue.

Future Work. In the future, we intend to extend the subset of xUML covered in our translation. In particular, we would like to include synchronous calls, which are used in the more elaborate xUML models of railways interlockings that we have been presented with. We would also like to extend our approach to specifying safety properties in UML to include specification of temporal properties.

Moreover, we would like to give a formal operational semantics of our xUML models, so that we can make formal statements about the generic properties of models, as well as a formal comparison of different alternative approaches to the xUML semantics. In order to enhance the scalability of our verification techniques, we will continue our efforts to adopt compositional techniques as well as counterexample-guided abstraction refinement.

Acknowledgements. The authors thank Louis Rose for his helpful comments on an earlier draft of this paper.

Funding. This research is partially funded by the European Commission (EC), as a grant to the FP7 project INESS, grant agreement no. 218575. Any opinions, findings and conclusions or recommendations expressed in this material are those

of the authors and do not necessarily reflect the views of either the EC or the INESS consortium.

References

1. Abrial, J.-R.: *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, Cambridge (2010)
2. Alur, R., Yannakakis, M.: Model checking of hierarchical state machines. *ACM Transactions on Programming Languages and Systems* 23(3), 273–303 (2001)
3. ter Beek, M.H., Fantechi, A., Gnesi, S., Mazzanti, F.: A state/event-based model-checking approach for the analysis of abstract system properties. *Science of Computer Programming* 76(2), 119–135 (2011)
4. Behrmann, G., Larsen, K.G., Andersen, H.R., Hulgaard, H., Lind-Nielsen, J.: Verification of hierarchical state/event systems using reusability and compositionality. *Formal Methods in System Design* 21(2), 225–244 (2002)
5. Bergstra, J.A., Klop, J.W.: *Process algebra for synchronous communication*. *Information and Control* 60(1-3), 109–137 (1984)
6. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. *Advances in Computers* 58, 118–149 (2003)
7. Blom, J., Hessel, A., Jonsson, B., Pettersson, P.: Specifying and generating test cases using observer automata. In: Grabowski, J., Nielsen, B. (eds.) *FATES 2004*. LNCS, vol. 3395, pp. 125–139. Springer, Heidelberg (2005)
8. Blom, S., van de Pol, J., Weber, M.: LTSmin: Distributed and symbolic reachability. In: Touili, T., Cook, B., Jackson, P. (eds.) *CAV 2010*. LNCS, vol. 6174, pp. 354–359. Springer, Heidelberg (2010)
9. Ciardo, G., Lüttgen, G., Miner, A.S.: Exploiting interleaving semantics in symbolic state-space generation. *Formal Methods in System Design* 31(1), 63–100 (2007)
10. Cimatti, A., Giunchiglia, F., Mongardi, G., Romano, D., Torielli, F., Traverso, P.: Formal verification of a railway interlocking system using model checking. *Formal Aspects of Computing* 10(4), 361–380 (1998)
11. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM* 50(5), 752–794 (2003)
12. Damm, W., Josko, B., Pnueli, A., Votintseva, A.: A discrete-time UML semantics for concurrency and communication in safety-critical applications. *Science of Computer Programming* 55, 81–155 (2005)
13. Eriksson, L.-H.: Specifying railway interlocking requirements for practical use. In: *Proceedings of the 15th International Conference on Computer Safety, Reliability and Security (SAFECOMP 1996)*. Springer, Heidelberg (1996)
14. Fokkink, W.: Safety criteria for the vital processor interlocking at Hoorn-Kersenboogerd. In: *Computers in Railways V (COMPRAIL 1996)*. *Railway Systems and Management*, vol. I (1996)
15. Formal Systems (Europe) Ltd. *Failures-divergence refinement: FDR2 User Manual* (2010)
16. Gavel, H., Lang, F., Mateescu, R., Serwe, W.: CADP 2010: A toolbox for the construction and analysis of distributed processes. In: Abdulla, P.A., Leino, K.R.M. (eds.) *TACAS 2011*. LNCS, vol. 6605, pp. 372–387. Springer, Heidelberg (2011)
17. Geilen, M.: On the construction of monitors for temporal logic properties. *Electr. Notes in Theor. Comp. Sci.* 55(2) (2001)

18. Ghazel, M., Toguyéni, A., Yim, P.: State observer for DES under partial observation with timed petri nets. *Discrete Event Dynamic Systems* 19(2), 137–165 (2009)
19. Gnesi, S., Latella, D., Lenzini, G., Abbaneo, C., Amendola, A.M., Marmo, P.: An automatic SPIN validation of a safety critical railway control system. In: *Proceedings of the 2000 Int. Conf. on Dependable Systems and Networks*, pp. 119–124. IEEE Computer Society, Washington, DC, USA (2000)
20. Graw, G., Herrmann, P.: Transformation and verification of Executable UML models. In: *Proceedings of the Workshop on the Compositional Verification of UML Models*. *Electr. Notes in Theor. Comp. Sci*, vol. 101, pp. 3–24 (2004)
21. Groote, J.F., Mathijssen, A., Reniers, M.A., Usenko, Y.S., van Weerdenburg, M.: The formal specification language mCRL2. In: *Methods for Modelling Software Systems*. *Dagstuhl Seminar Proceedings*, vol. 06351 (2007)
22. Hansen, H.H., Ketema, J., Luttkik, B., Mousavi, M.R., van de Pol, J.: Towards model checking Executable UML specifications in mCRL2. *Innovations in Systems and Software Engineering* 6(1-2), 83–90 (2010)
23. Heidenreich, F., Johannes, J., Karol, S., Seifert, M., Wende, C.: Derivation and refinement of textual syntax for models. In: Paige, R.F., Hartman, A., Rensink, A. (eds.) *ECMDA-FA 2009*. LNCS, vol. 5562, pp. 114–129. Springer, Heidelberg (2009), <http://www.emftext.org> (last visit: July 4, 2011)
24. Hoare, T.: *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs (1985)
25. Holzmann, G.J.: *The SPIN Model Checker*. Addison-Wesley, Reading (2003)
26. ISO/IEC. Enhancements to Lotos (E-Lotos), International Standard 15437:2001 (2001)
27. ISO/IEEE. ISO/IEEE 11073-20601: Health informatics — personal health device communication — Part 20601: Application profile — optimized exchange protocol (April 2010)
28. Keiren, J.: *Modelling session setup of IEEE Std 11073-20601 (2011), Personal communication*
29. KnowGravity. *Cassandra/xUML User’s Guide* (2008)
30. Kolovos, D.: *An Extensible Platform for Specification of Integrated Languages for Model Management*. PhD thesis, University of York, United Kingdom (2009), <http://www.eclipse.org/gmt/epsilon/> (last visit: July 4, 2011)
31. Kolovos, D., Rose, L., Paige, R.: *The Epsilon Book*, <http://www.eclipse.org/gmt/epsilon/doc/book/> (last visit: July 4, (2011)
32. Lafortune, S., Teneketzis, D., Sampath, M., Sengupta, R., Sinnamohideen, K.: Failure diagnosis of dynamic systems: an approach based on discrete event systems. In: *Proceedings of the American Control Conference*, vol. 3, pp. 2058–2071 (2001)
33. Lind-Nielsen, J., Andersen, H.R., Hulgaard, H., Behrmann, G., Kristoffersen, K.J., Larsen, K.G.: Verification of large state/event systems using compositionality and dependency analysis. *Formal Methods in System Design* 18(1), 5–23 (2001)
34. Mekki, A., Ghazel, M., Toguyeni, A.: Time-constrained systems validation using MDA model transformation. A railway case study. In: *Proceedings of the 8th International Conference of Modeling and Simulation, MOSIM 2010* (2010)
35. Mellor, S.J., Balcer, M.: *Executable UML: A foundation for model-driven architecture*. Addison-Wesley, Reading (2002)
36. Object Management Group. *OMG Unified Modeling Language Superstructure Version 2.2* (February 2009)
37. Papyrus Developers. *Papyrus: Open source tool for graphical UML2 modelling*, <http://www.papyrusuml.org> (last visit: July 4, 2011)

38. Schneider, F.B.: Enforceable security policies. *ACM Transactions on Information and Systems Security* 3(1), 30–50 (2000)
39. Sheeran, M., Stålmarck, G.: A tutorial on stålmarck’s proof procedure for propositional logic. In: Gopalakrishnan, G.C., Windley, P. (eds.) *FMCAD 1998*. LNCS, vol. 1522, pp. 82–99. Springer, Heidelberg (1998)
40. Sighireanu, M.: *LOTOS NT user’s manual*. Technical report, INRIA Rhône-Alpes/VASY (2008)
41. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: *EMF: Eclipse Modeling Framework*. Addison-Wesley Professional, Boston (2008), <http://www.eclipse.org/modeling/emf/> (last visit: July 4, 2011)
42. Turner, E., Treharne, H., Schneider, S., Evans, N.: Automatic generation of CSP || B skeletons from xUML models. In: Fitzgerald, J.S., Haxthausen, A.E., Yenigun, H. (eds.) *ICTAC 2008*. LNCS, vol. 5160, pp. 364–379. Springer, Heidelberg (2008)
43. Winter, K., Robinson, N.J.: Modelling large railway interlockings and model checking small ones. In: *ACSC 2003: Proceedings of the 26th Australasian Comp. Sci. Conference*, pp. 309–316. Australian Computer Society, Inc. (2003)
44. Yeung, W.L., Leung, K.R.P.H., Wang, J., Dong, W.: Improvements towards formalizing UML state diagrams in CSP. In: *Proceedings of the 12th Asia-Pacific Software Engineering Conference (APSEC 2005)*. IEEE Computer Society, Los Alamitos (2005)

Verification of UML Models by Translation to UML-B

Colin Snook, Vitaly Savicks, and Michael Butler

University of Southampton

Abstract. UML-B is a ‘UML like’ notation based on the Event-B formalism which allows models to be progressively detailed through refinements that are proven to be consistent and to satisfy safety invariants using the Rodin platform and its automatic proof tools. UML, on the other hand, encourages large models to be expressed in a single, detailed level and relies on simulation and model testing techniques for verification. The advantage of proof over model-testing is that the proof is valid for all instantiations of the model whereas a simulation must choose a typical instantiation. In the INESS project we take an extant UML model of a railway interlocking system and explore methodical ways to translate it into UML-B in such a way as to facilitate proof that the model satisfies certain safety properties which are expressed as invariants. We describe the translation attempted so far and insights that we have gained from attempting to prove a safety property. We propose some possible improvements to the translation which we believe will make the proof easier.

1 Introduction

The aim of the INESS project [1] is to develop specifications and associated material to assist in the development of a European common standard for railway interlocking systems. The role of the WP4 group within INESS is to develop ways to verify UML models of such interlocking systems. Other partners in WP4 are using model-checkers to verify a translation of the UML models. While this method benefits from a high degree of automation, the size of models that can be verified is limited by performance constraints and each instantiation of the model (i.e. interlocking layout) has to be verified separately. We are exploring the alternative approach of using theorem provers to verify (a translation of) the model. While this approach generally requires a higher degree of expert intervention depending on the size and complexity of the model, it does not suffer from state-explosion to the same degree and once completed is valid for all possible layouts that satisfy the constraints of the model. Proof however, can quickly become intractable in real-world problems. The key to proving something, therefore, is to abstract away from the details and prove much simpler properties which can then be used as lemmas in the proof at the more detailed level. Refinement is the process of introducing more detail into the model in order to move from the simpler abstract version to the detailed concrete level and

requires further proof to ensure that the concrete model is consistent with the abstraction. The refinement that we use can be broadly categorised into horizontal, superposition where more details are added without altering the representation of the abstract model, and vertical data refinement where the abstract model is replaced with a more complex version. The latter is, in general, a more powerful form of abstraction and involves more substantial proof to show the correspondence between the models. In fact, the intention of the refinement is generally given in the form of a ‘gluing invariant’ which specifies the relationship between the abstract variables and the concrete ones. The gluing invariant may also be used as a lemma in the proof of properties at the detailed level.

In this paper we report on an investigation into translating a UML model of an interlocking system into a formal refinement based notation so that safety properties can be formally proven to hold. We use a small given example of an interlocking model which is constructed in a style which is proposed by the INESS project for use in the railway industry sector and we select and prove one safety property from its requirements specification to demonstrate our method. We would normally recommend considering safety properties at the earliest possible stage [2] but this requires expertise in finding useful abstractions. Instead, we adopt a rather naive approach where our refinements are constructed mechanically from the UML structure without abstracting the main concepts involved in the safety property and we then attempt to add the safety property at a late stage. The purpose of taking this approach was to determine to what degree a mechanistic, low expertise, approach would succeed and where we would need to introduce more abstraction in order to succeed in the proof. Indeed, when we initially tackled the proof of the safety requirement, not only did we find it difficult, but also, we could not detect whether it should be provable or whether there was a problem in the model. For a second attempt, we introduced a more useful abstraction into the model which enabled us to detect some errors in the original UML source model and then prove the safety requirement in a more abstract form. We were then able to prove the more concrete form of the safety requirement using the abstract property.

Our goal in the INESS project is to develop techniques for using our methods to prove safety requirements in UML models of interlocking systems. As depicted schematically in Fig. 1, to do this we need to translate the UML model and its associated safety requirements into our notation where we can apply the automatic prover. The primary aim of this paper is to report on our ongoing work on INESS. However, we also see a more general contribution emerging which is a set of guidelines for building formal models from semi-formal entity relationship and state machine models (including UML models) so that the process is better defined and it is clearer where stronger abstractions will be needed than those that can be inferred mechanically.

The paper is structured as follows. In section 2 we describe the methods that we use. In section 3 we describe the source UML model that we were given to verify. In section 4 we describe our mechanistic approach to translating the UML model into a series of UML-B refinements. In section 5 we describe our

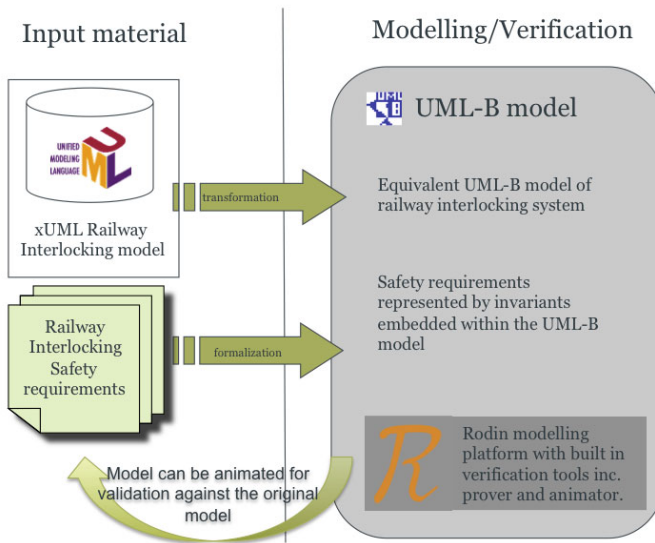


Fig. 1. Schematic block diagram illustrating the translation of the UML model and associated safety invariants into the UML-B notation

interpretation of a safety requirement and how the pursuit of a proof led us to revise our modelling approach and to uncover safety problems in the original model. Section 6 outlines how we intend to develop our approach in response to the findings reported here. Section 7 is the conclusion.

2 Background

The Unified Modelling Language (UML) [3] is a semi-formal diagrammatical modelling notation which has been adopted relatively widely throughout industry. UML includes several diagram notations which can be used for different aspects of a system and for different stages of the development process. The UML diagram notations that we are concerned with are ‘Class Diagrams’ which can be used to show the relationships between different kinds of entities and ‘State Diagrams’ which give a state oriented view of the behaviour of the classes. To a lesser extent we are also interested in ‘Use Case Diagrams’ and ‘Sequence Diagrams’ which are used together to illustrate the requirements of a system. UML is often interpreted flexibly and to some extent this is a strength, but it is sometimes criticised for having imprecise semantics which may lead to confusion and ambiguity. Some variants of UML have been developed which are constrained and more precisely defined. UML has no notion of refinement.

Event-B [4,5] is a state-oriented formal modelling language that was developed for modelling at the systems level. State is represented by typed variables and

spontaneous transitions (guarded events) occur to alter the state. A central concept of Event-B is refinement, where more detailed state is added and this reveals more detailed events. The Rodin platform has been developed as a formal modelling environment for Event-B and includes a static checker and a prover. The Rodin platform [6,7] is designed to be extensible and many plug-in's are available which extend the Event-B language or provide additional tools for model development, verification and validation.

We base our approach upon UML-B [8,9], Event-B and the Rodin platform. UML-B is a visual front-end for the Event-B notation and includes a state machine diagram editor. Tool support for UML-B is provided by a plug-in to the Rodin platform. State machines may be refined by adding nested state machines [10] and can be animated via a plug-in [11] that utilises the ProB [12] model checker and animator. The state machine refinement supported by UML-B allows the model to be progressively developed in stages. This improves understanding, and hence validity, as features can be laid down in small steps while the Rodin provers ensure consistency. Invariants can be added to states (i.e. the state being active becomes an implicit antecedent for the property) providing another mechanism to clearly state our understanding of the model with further consistency checking via the Rodin provers. One use for invariants is to express safety properties that we believe the model satisfies so that we can prove that this is indeed the case. Animation of the state machine diagrams allows us to test that they behave as we expected.

3 UML Model of Interlocking

For the purpose of investigating and demonstrating how to apply our methods in this domain, we were given a small UML model of an interlocking system, called *micro2010* [13], which had been constructed in a style that is typical of the full scale models that are expected to be produced during or after the INESS project. The model is constructed using the Artisan UML tool [14] in a variant of the UML called xUML [15] which can be executed in a simulation environment called Cassandra [16]. xUML has a precise but easy to read action and constraint language which is interpreted by the Cassandra simulation tool. The advantage of this model is that it has an operational semantics as defined by the Cassandra simulation.

The *micro2010* UML model is based on a structure of classes as shown in Fig 2. The class *HAL* represents the control of a physical device in the system. Hence the class structure making up the modelled interlocking system controller mirrors the components in the physical track layout. There are subclasses for *signal*, *track* and *point*. The class, *route*, represents a concept used in railway interlocking systems where a collection of track layout components constitute a path through the layout that a train may wish to follow. The safety of the system is based on a route being requested and subsequently allocated provided that all the components in that route are free and not allocated for use in any other route. The route class therefore has several associations with the logical

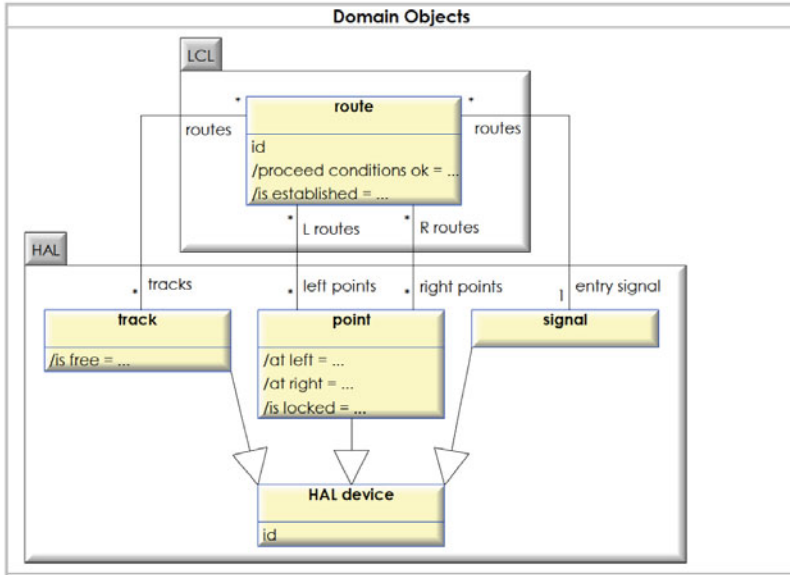


Fig. 2. UML Class Diagram showing structure in the interlocking model

device classes which identify the components involved in that route. Note that, for verification by proof, we do not need to specify the values of these classes and associations, the proof is valid for any valid instantiation, whereas, for model checking or simulation an example configuration is configured or generated by instantiating these classes and giving values to all the associations.

The classes often own derived attributes which represent a boolean condition over properties of the class and its associated classes. Derived attributes (designated by a preceding slash /) do not represent any new state, they are a shorthand way of expressing a boolean condition over variables that are elsewhere in the model. For example, the *point* class has a derived attribute, */is_locked*, which is true if and only if that point belongs to a route that is currently established. There is no actual variable attached to */is_locked*, so it can never be assigned to in an action but it may appear in guards as a shorthand for its definition.

The behaviour of each class is specified by a state machine diagram. Examples of two such state machines are shown in Figs 3 and 4. The state machines are composed in a hierarchical manner with some states containing sub-states. The behaviour of the system is specified in terms of guarded transitions that may fire when the system is in their source state. For convenience, guards are usually specified using a derived attribute (e.g. *moveLeft[not /isLocked]* in Fig 3). There are several different ways by which a transition may be triggered. Transitions stereotyped *ic* are triggered by a send action in another transition (e.g. *moveLeft* in Fig 3). Transitions stereotyped *dv* are triggered by external events in the environment (e.g. *atLeft* in Fig 3). Transitions named with the keyword ‘after(t)’ are triggered spontaneously after the time t since the source

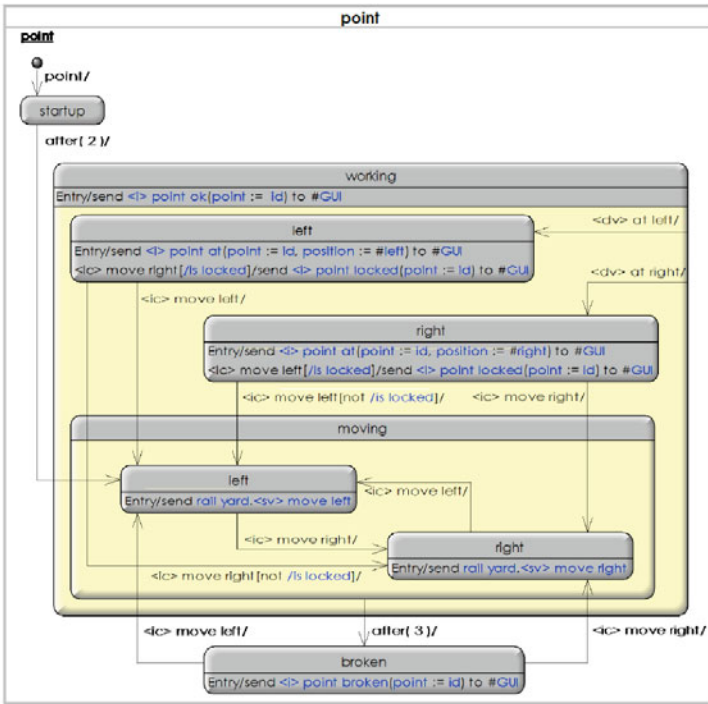


Fig. 3. UML State Machine of Points

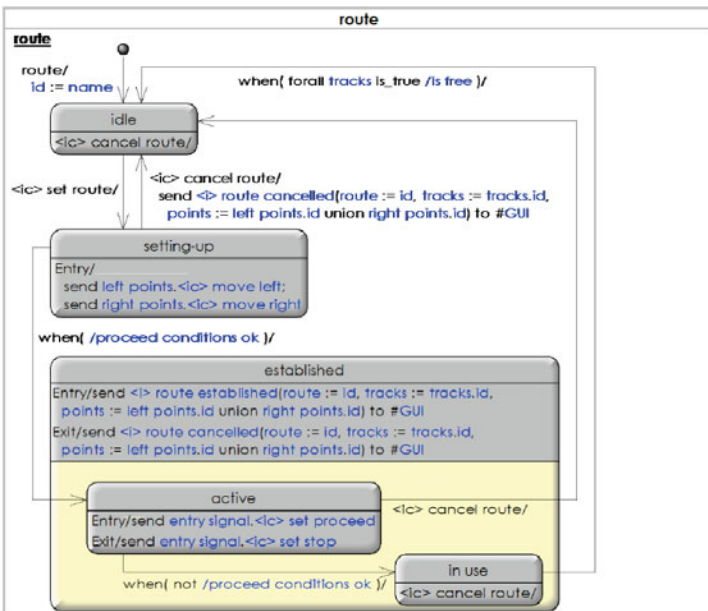


Fig. 4. UML State Machine of Routes

state was entered (e.g. *after(3)* in Fig 3). Transitions named with the keyword, ‘when’, are triggered spontaneously when their guard condition is true (e.g. *when(!/proceed-conditions-ok)* in Fig 4).

4 Translation to UML-B

UML-B provides equivalent modelling diagram notations to those used in the UML model (i.e. Class and State-machine) but with minor syntactic and significant semantic differences. In general UML-B is less flexible because it is strongly founded on the Event-B formalisation, but in most cases there are no significant problems in translating the diagrams. We also introduced some new features into UML-B in order to assist the translation. However, the semantic differences in transitions deserves some attention. Due to their correspondence with Event-B events, the only mechanism for triggering transitions in UML-B is spontaneous triggering when the transition guard is true (‘when’ transitions). The other transition triggering methods used in the UML model are handled as follows.

Externally triggered transitions are modelled using spontaneous triggering in the same way as ‘when’ transitions. Since we do not explicitly model the environment, the transition is considered to implicitly represent the response to an event occurring in the environment.

Timed transitions are modelled using spontaneous triggering in the same way as ‘when’ transitions. Since we do not explicitly model time, the transition is considered to implicitly represent the response to a time limit being reached. Since the properties we wish to verify only concern event ordering, this is sufficient for our purposes provided that there are no cases where two timing events compete from the same system state.

Internally triggered transitions are represented by modelling a message passing system. A base class *buffer_owner* is inherited by all other classes and provides an attribute which is a set of messages received by an object in that class. A transition that needs to be internally triggered waits for an instance of its trigger message to be received by its owning object and removes that message as it fires. In a UML-B refinement, new transitions may only modify variables that have been introduced in that refinement and are not allowed to alter the variables introduced in previous refinement levels. This means that we can not alter the buffer in any subsequent refinement levels after we first introduce the buffer. To avoid this problem we provide an event that non-deterministically modifies the buffer attribute so that transitions introduced in subsequent refinements may refine this behaviour by sending trigger messages to other objects and removing their own trigger messages. Note that although this mechanism does not impose any ordering on triggering within an object and does not allow for multiple triggering of the same transition it is sufficient for the safety properties being verified.

As in Event-B, refinement is a key concept within UML-B. Even without introducing a safety property, there are significant proof obligations concerning the internal consistency and well-definedness of the model and these would be

difficult to handle if the model was introduced in a single stage without refinement. Therefore, we need to build the UML-B model in several refinement stages. Since this is a research experiment aimed at finding a method that can be applied to bigger and more general UML models of interlocking systems, we also require that the method of introducing refinements is methodological and does not rely on high expertise in refinement and abstraction. Three potential methods for introducing refinement are considered.

- a) Class inheritance has some analogy with refinement. Classes higher up the inheritance structure can be introduced and modelled without knowledge of subclasses. These inherited classes have features that are common to their subclasses which can be modelled in an abstract level before the subclasses are introduced in a subsequent refinement. In this case the micro2010 model contains one inherited class, *HAL* which does not contain any interesting behaviour. Therefore we do not use this technique in this translation.
- b) Class associations and behaviour can be examined for dependency and used to determine a priority ordering for introducing classes. It is usually the case that the classes exhibit a hierarchy in the way that they control each other with one class responding to instructions from another (via the internal triggering mechanism). The associations give a clue to this hierarchy because internal triggering occurs between instances that are linked by associations. In the class diagram of the micro2010 model (Fig. 2) observe that the *track*, *signal* and *point* classes are not connected by association whereas the *route* class connects with all three. It is apparent that the *track*, *signal* and *point* classes are independent and can be introduced in any order whereas the *route* class is dependent on the others and must be introduced last. This is confirmed by examining the internal triggering where the *route* class is the only one that sends triggers while the other classes are only responding to them.
- c) State machine hierarchy can be used to introduce the full behaviour in stages following the hierarchy of nesting within the class' state machine. Nested state machines only elaborate the behaviour of the parent state machine which can be constructed in a way that is valid without the nested one. We use this method to introduce the *point* class in three refinements and again to introduce the *route* class in two refinements.

These 3 methods should be used in the order shown since this will result in an appropriate ordering with respect to class dependencies. We also suggest using the 3 methods in a 'depth first' manner. That is, the highest (most abstract) level classes in the inheritance structure should be introduced first and fully refined using methods b and c before the next level in the inheritance structure is dealt with. Similarly the classes introduced by analysing the associations should be fully refined by method c before the next priority according to associations is dealt with.

There is no facility in UML-B for representing derived attributes. Therefore derived attributes are fully expanded with their definitions wherever they are used and only introduced at a refinement level where all the features required

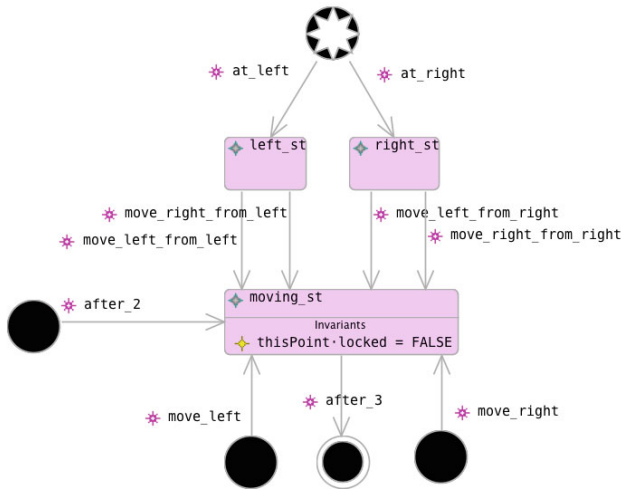


Fig. 5. UML-B State Machine Diagram showing the behaviour of points in the working state

by their definitions have been added to the model. However, our experience of proving (described in chapter 5) has led us to the conclusion that there needs to be more abstraction of features of the model. We are now re-assessing whether these derived attributes represent useful abstractions that should be introduced in early refinements for the purpose of proving important properties.

The refinement steps of the UML-B version of the railway interlocking model can be summarised as follows: (The application of the methods outlined above is shown for each refinement).

- m0** Introduces the message passing mechanism used for internal triggering of transitions. (No method - prior to translation).
- m1** Introduces the *signal* class and its state-machine describing its behaviour which sets the signal to *proceed* or *stop* in response to internal triggers. (Method b).
- m2** Introduces the *track* class and its state-machine describing its behaviour which sets the track to *free* or *occupied* in response to internal triggers. (Method b).
- m3** Introduces the *point* class and its first level state-machine describing its behaviour to go to the *broken* state if a command is not achieved within a time limit and to recover if another internal trigger is received. (Method b).
- m4** Elaborates the behaviour of the *point* class when it is in the *working_st* state by adding the nested state-machine shown in Fig. 5. (Method c).
- m5** Elaborates the behaviour of the *point* class when it is in the *moving_st* state by adding the nested state-machine shown in Fig. 6. (Method c).
- m6** Introduces the *route* class and its first level state-machine shown in Fig. 7. (Method b).

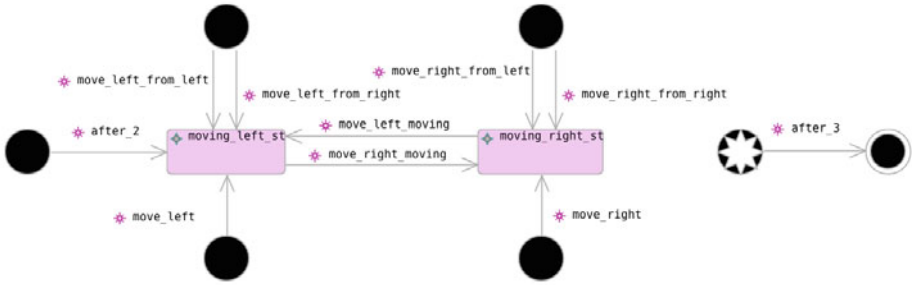


Fig. 6. UML-B State Machine Diagram showing the behaviour of points in the moving state

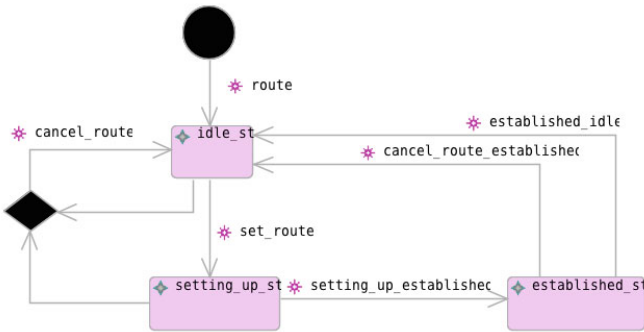


Fig. 7. UML-B State Machine Diagram showing the behaviour of routes

m7 Elaborates the behaviour of the *route* class when it is in the *established_st* state by adding a nested state-machine. (Method c).

m8 Adds the safety invariant to check that points that are in an established route do not move. (No method - add invariant).

In order to reason about the UML-B model and, in particular, to discuss the invariants, we need a textual representation of the value of a state machine. This is provided by the UML-B toolset since it automatically translates the diagrams into an equivalent Event-B model for verification purposes. The tool translates each class into a set with the same name as the class, and each state machine to a variable using the name of the state machine. (As a convention we have named the state machines after their parent with *”_sm”* appended). The state machine variable is a function from the set representing the class to an enumeration of the states in the state machine. The value of a state machine for a particular instance of a class is therefore available by function application. Associations are constant relations between the class sets and values can be accessed by relational image. Hence we can refer to the left points of a route, *r*, as *left_points[{r}]* and a point, *p*, being in the left position as *point_sm(p) = left_st*.

5 Proving the Safety Invariant

Once the UML-B model is developed throughout its levels of refinement, we can turn our attention to verifying that it satisfies the safety requirements. Up to this stage, until we introduce some safety requirements, the proof only concerns:

- a) well-definedness (e.g. that where a function application is used, the function is defined for that value),
- b) typing (i.e. that an assignment doesn't contravene a defined sub-range of a variable's basic type) and
- c) simulation (i.e. that the the principles of refinement are observed).

These proof obligations, which are mostly discharged automatically by the Rodin provers¹, ensure that the model is constructed correctly in a consistent manner but do not prove anything about how the model behaves.

A more interesting and challenging use of proof is to introduce some invariant property which we require to hold at any time in the model. These invariant properties are ideal for expressing safety requirements. The micro2010 UML model contains four safety requirements which are stated in natural language in the documentation. We choose one of these safety requirements for the purpose of illustrating our method.

SR1: A point that is locked by an established route shall never move.

The safety requirement is worded with a little redundancy since, by definition, a point is locked by establishing a route to which it belongs. Since we have expanded away all derived attributes, such as */is_locked*, in our UML-B model, we re-state the safety invariant as follows:

SR1': A point that belongs to the left points or right points of an established route shall never move.

We can now translate this into an invariant using the features of our UML-B model. When translated to plain Event-B this state invariant becomes::

SR1 : $\forall r, p. r \in \text{route} \wedge \text{route_sm}(r) = \text{established_st} \wedge p \in (\text{left_points}\{\{r\}\} \cup \text{right_points}\{\{r\}\}) \Rightarrow (p \in \text{dom}(\text{working_sm}) \Rightarrow \text{working_sm}(p) \neq \text{moving_st})$

SR1 is read as follows: For all p and r, where r is a route which is in the state *established_st* and p belongs to the union of the *left_points* and *right_points* of r then (if p is in the domain of the state machine, *working_sm* then) p is not in the state *moving_st*. Note that the condition in brackets is not a logical necessity since if it is not true then the point is not even in the super state of *moving_st*, however, it is included so that the prover can discharge a related proof obligation that the function application, *working_sm(p)*, is well formed.

¹ Some theorems were added at the first refinement to help the provers discharge POs about the refinement of buffer assignments.

We were unable to prove this safety invariant and suspected the original UML model to be unsafe. Often when this is the case, examination of the proof obligation helps understanding the problem and suggests a correction in the model. However, in this case it is not obvious why the proof obligation is not provable and how the model can be fixed. The difficulty stems from the late arrival of the concept of routes and the consequent lack of abstract representations of the concepts involved in the safety invariant. In fact, the original UML model introduces the abstract concept of locking (albeit for different reasons) which we have discarded due to difficulty in translation. To improve our model we re-worked it from refinement level m_4 where *point* is introduced and at this stage we model the derived attribute *is_locked* as a boolean attribute of the *point* class. This allows us to express, at an earlier stage, the fact that points must not move while locked, even before specifying the real meaning of locked. At this stage we introduce some non-deterministic alteration of the locked attributes which are later refined into the behaviour of routes. We introduce an invariant into the *moving_st* state of the *point* class to express the constraint that the point should not be locked when it is moving. In doing so we prove an abstract equivalent of the safety invariant SR1, namely:

SR1a: A point that is moving is not locked.

The invariant can be seen in the state *moving_st* of Fig 5. When translated to plain Event-B this state invariant becomes:

SR1a : $\forall thisPoint. ((thisPoint \in point) \Rightarrow ((point_sm(thisPoint) = working_st) \Rightarrow ((working_sm(thisPoint) = moving_st) \Rightarrow (locked(thisPoint) = FALSE))))$

The invariant in the diagram is translated from dot notation to function application to obtain $locked(thisPoint) = FALSE$. The contextual position of the invariant in the sub-state *moving_st* of the state *working_st* of a state machine belonging to the class, *point*, gives rise to the chain of antecedents.

In attempting to prove this we realised that the model has unguarded transitions to *moving_st* which are triggered when a point is requested to move to the position it is already in (transitions *move_left_from_left* and *move_right_from_right*). Perhaps this was not considered by the UML modellers to be a genuine move for the purpose of safety but, if this is the case, it is not possible to (formally) distinguish safety in the given model meaning that we have been given an impossible task. To investigate further we corrected the model by adding guards to prevent these transitions when the point is locked allowing us to prove the abstract safety invariant SR1a.

Having established that locked points do not move, to complete the proof of the safety invariant, we need to prove that the points in an established route are always locked. (This is the gluing invariant corresponding to the data refinement that replaces *is_locked* with membership of an established route). That is:

GL1 : $\forall r, p. r \in route \wedge route_sm(r) = established_st \wedge p \in (left_points[\{r\}] \cup right_points[\{r\}]) \Rightarrow locked(p) = TRUE$

Given the abstract invariant, SR1a and the gluing invariant, GL1, the original safety invariant, SR1, is discharged easily by the Rodin provers. In fact, since it follows directly from SR1a and GL1 we can make SR1 a theorem so that it only needs to be proved once.²

However, the gluing invariant still can not be proved for the transitions, *cancel_route* and *established_idle*, that reset locked to false (or to use the concrete representation, release an established route). Upon examination, we realise that the model does not prevent two conflicting routes (that share common points) from being established concurrently (so that a point is locked by two routes at the same time). In this case, the gluing invariant is violated when either one of the routes is cancelled and unlocks its points. The location of the mistake is in another derived attribute of the UML model, *proceed_conditions_ok*, which is defined as ‘all *left_points* of the route are *at_left*, all *right_points* of the route are *at_right* and all *tracks* of the route are *free*’. Hence the proceed conditions defined in the source model prevent a route from being established when its points are not in the required positions but not if another route has already established and locked those points in the correct positions. Strictly speaking, the safety requirement we are working on does not specify anything about conflicting routes, it was an assumption we made in our gluing invariant. We could design an alternative, provable, gluing invariant which allows for conflicting routes, but since the micro2010 UML model also contains a use-case that indicates that a ‘set route’ request should not succeed if one of its points is locked by a route, we deduce that there is a mistake. To correct it we add a guard to the transition, *setting_up_established*, in the route state machine Fig. 7 of refinement level *m6*. The guard ensures that none of the points that are used in the route are already locked:

Guard4 : $locked[left_points[\{self\}] \cup right_points[\{self\}]] = \{FALSE\}$

The guard uses the contextual instance, *self*, of the class, *route* (i.e. *self* is the route being set up by the transition *setting_up_established*). The guard can be read as, the only value of *locked* for all the *left_points* and *right_points* of *self*, is *FALSE*.

Note that this is the same transition that locks the points used by the route. It is important that this is done in one (atomic) transition to a) ensure that the guard remains true when the locks are set and b) so that the derived attribute *is_locked* is refined by its definition.³

After adding this guard we are able to prove the gluing invariant for the two transitions that render a route un-established, i.e. *cancel_route* and *established_idle*. This completes the proof of the safety invariant SR1.

² In Event-B, and hence UML-B, to prove an invariant it is necessary to prove that every event results in a state that satisfies the invariant whereas a theorem is deduced from the other invariants and theorems.

³ Although, for clarity, we have retained *is_locked*, we would like the option to remove it in a later refinement and our proof of SR1 relies on the data refinement where it is replaced by its definition.

The statistics for proofs are given in the following table. The automatic provers are configurable so that the user can choose which provers to try and specify time-out values. These figures represent a configuration with a meta-prover (called *Relevance Filter*) enabled. With this meta prover enabled, before tackling the safety requirement in m8, only one PO (from m6) requires interactive proof. All of the POs relating to the abstract safety invariant in m4 are discharged automatically. In m8, three PO's required interactive proof. These are the proof of the intermediate state invariant when a route is established and the two (identical) proofs that the transitions that un-establish a route satisfy the gluing invariant.

Refinement	Proofs	Automatic	Interactive
m0	7	7	0
m1	11	11	0
m2	18	18	0
m3	15	15	0
m4	88	88	0
m5	34	34	0
m6	34	33	1
m7	26	26	0
m8	16	13	3

It is interesting to know where the effort lies in carrying out this verification work. The effort in actually carrying out an interactive proof is minor. Now that the model has been constructed and corrected, we could conduct the interactive proofs within an hour at most. Similarly, discovering and correcting errors in the model is not particularly time consuming because the proof obligations quickly lead one to the problem. However, the iterative process of attempting the proofs, running into difficulties and improving the modelling techniques to make the model more amenable to proof, took several weeks of effort.

6 Future Work

As part of our role in the INESS project we will continue to prove other safety requirements in the micro2010 model. We expect to find that similar techniques of introducing more abstraction are necessary and hope this will lead to a general method. Hence we will investigate in more detail how to methodically generate abstract concepts from UML derived attributes in a way that helps us prove safety invariants. We will continue this investigation using more extensive examples in order to test the generality of the method.

We are also interested in developing guidelines for constructing a UML-B model in refinements from a UML model that has none. For this purpose, the micro2010 model has given us some initial ideas but is somewhat limited. We plan to investigate these guidelines using some more extensive examples of interlocking models that are also available as part of the INESS project.

Another approach that we have previously investigated [2] is to start by modelling the most abstract model that is needed in order to demonstrate the safety requirements. It may then be possible to develop the full translated model as a refinement of the safety requirements, hence showing that the model is safe by construction. We would like to investigate this alternative approach in comparison to that presented here.

7 Conclusion

We have investigated a mechanistic approach to translation of a UML model into a series of UML-B refinements and the introduction of safety invariants at a final stage. We found that this approach has limitations because the refinement is based on the structure of the UML model resulting in a superposition approach to refinement (often called horizontal refinement). The safety invariant to be proved becomes too complicated to a) find a proof, and b) to find why it is not provable in the model. We deduce that the translation does not provide enough abstraction of concept (often called vertical refinement) and suggest that derived attributes of the UML source model may provide a clue to introducing this methodically through the translation. When we introduced a vertical refinement using the derived attribute *is.locked* we were able to find and rectify two problems in the model and subsequently proved the safety invariant with relative ease using the abstract version of it and the gluing invariant of the refinement.

References

1. INESS (2010), <http://www.iness.eu/>
2. Snook, C.: Specifying Safety Requirements for a Railway Interlocking System. Dagstuhl Seminar on Refinement based methods for the construction of dependable systems (2009)
3. Rumbaugh, J., Jacobson, I., Booch, G.: Unified Modeling Language Reference Manual, 2nd edn. Addison-Wesley Object Technology. Addison-Wesley Professional, Reading (2004)
4. Metayer, C., Abrial, J.R., Voisin, L.: Event-B Language. Rodin deliverable 3.2, EU Project IST-511599 -RODIN (May 2005)
5. Abrial, J.R.: Modelling in Event-B: System and Software Engineering. Cambridge University Press, Cambridge (2010)
6. Abrial, J.R., Butler, M., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: an open toolset for modelling and reasoning in Event-B. International Journal on Software Tools for Technology Transfer (STTT) 12(6), 447–466 (2010)
7. Abrial, J.R., Butler, M., Hallerstede, S., Voisin, L.: An Open Extensible Tool Environment for Event-B. In: Liu, Z., Kleinberg, R.D. (eds.) ICFEM 2006. LNCS, vol. 4260, pp. 588–605. Springer, Heidelberg (2006)
8. Snook, C., Butler, M.: UML-B: Formal modeling and design aided by UML. ACM Trans. Softw. Eng. Methodol. 15(1), 92–122 (2006)
9. Snook, C., Butler, M.: UML-B and Event-B: an integration of languages and tools. In: The IASTED International Conference on Software Engineering SE 2008 (February 2008)

10. Said, M., Butler, M., Snook, C.: Language and Tool Support for Class and State Machine Refinement in UML-B. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 579–595. Springer, Heidelberg (2009)
11. Savicks, V., Snook, C., Butler, M.: Animation of UML-B Statemachines. Technical Report (<http://eprints.ecs.soton.ac.uk/18261/1/TBFMsmAnim.pdf>) and presented at Rodin User and Developer Workshop (2010)
12. Leuschel, M., Butler, M.: ProB: A model checker for B. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 855–874. Springer, Heidelberg (2003)
13. Schacher, M.: Micro interlocking 2010. Know Gravity Inc (2010), <http://knowgravity.com>
14. ArtisanStudio (2010), <http://www.atego.com/products/artisan-studio/>
15. Mellor, S., Balcer, M.: Executable UML: A foundation for model-driven architecture. Addison-Wesley, Reading (2002)
16. Cassandra (2010), <http://www.knowgravity.com/eng/value/cassandra.htm>

Towards the UML-Based Formal Verification of Timed Systems

Luciano Baresi, Angelo Morzenti, Alfredo Motta, and Matteo Rossi

Politecnico di Milano
Dipartimento di Elettronica e Informazione, Deep-SE Group
Via Golgi 42 – 20133 Milano, Italy
{baresi,morzenti,motta,rossi}@elet.polimi.it

Abstract. This paper presents the approach to the formal verification of UML-based models of timed systems developed in the MADES project. The approach differs from many current ones in that it aims at (i) being inclusive in the range of diagrams considered when producing the formal model, and (ii) adhering to the UML notation as much as possible. The metric temporal logic-based semantics developed in the project is presented through an example system.

1 Introduction

UML, along with its dialects and profiles, is a widely utilized graphical, design notation. Despite the vast adoption, users only tend to agree on the interpretation of few well-known concepts, while the actual behavior of many parts of the notation is left open. The concrete syntax of the language is very rich, and offers alternatives to model the same concepts, but its semantics is only defined informally and imprecisely. This is enough if we think of UML as a pure modeling notation; it is not acceptable when one aims to detailed descriptions of the system-to-be, neither is it suitable for automated analysis and for deriving implementations that go beyond the frame of some classes.

In contrast, formal methods and tools (e.g., UPPAAL¹ or Alloy²), which would provide sophisticated analysis capabilities, have often demonstrated their inability to attract the masses: the required mathematical background hampers their adoption and many users privilege the “simplicity” of UML-like notations rather than more formal means. Many proposals (e.g., [10,11,18]) have already tried to bridge the gap between the two domains by attempting to provide (parts of) UML with a precise (possibly formal) semantics. The idea is to keep the positive aspects from both fields and provide the user with a well-known modeling notation, suitably augmented with a formal semantics behind the scene.

If we think of UML as design means for a well-known programming language (e.g., Java), the subset of the notation usually considered is very limited (mainly just class diagrams), and the actual semantics is assumed to be the same as

¹ <http://www.uppaal.org>

² <http://alloy.mit.edu/>

the one of the target language. Petri nets have been widely used to explain the dynamic behavior of UML activity and state diagrams for years [19,12], but the immediate explosion of the resulting nets, along with the inability to easily distinguish between types (classes) and instances, hampered their adoption as underlying formal representation for UML models.

The heterogeneity and overlapping of the different modeling elements, the wide spectrum of the notation, and also the size of the resulting formal specifications, which are often too big for analysis, play against complete formalizations. A coherent, consistent, and complete formal semantics is a very complex and heavy task; things become even more complex when one considers some special-purpose extensions defined for particular domains.

These limitations have been our motivations, and challenge, for ascribing a formal dynamic semantics to a particular UML-based notation for timed systems, the MADES modeling notation [1]. This language borrows many concepts from SysML [15] and MARTE [14], but our interest is mainly in the timing aspects: we are interested in the *clocks* provided by MARTE, and in the UML diagrams that show time-related behaviors.

Being well aware of the difficulties inherent to the task, we decided to concentrate on a complete subset of the notation, which we called *verification* notation. This is limited with respect to the general modeling notation, but selected elements and diagrams are able to cover all the important aspects of a complete MADES model:

- The static parts of a system are covered through class diagrams. These diagrams, which can also be adopted to render components and objects, are used to define the terms (the alphabet) of the specification.
- The dynamic aspects and behavior of the different parts are rendered through:
 - (a) State diagrams (and thus also activity diagrams), used to model the behavior of the different elements (components),
 - (b) Sequence diagrams, used to model the “local” interactions among the different elements of the system, and
 - (c) Interaction overview diagrams, used to relate different sequence diagrams. Sequence diagrams are adopted to describe limited scenarios that define how the system reacts to some particular conditions and/or inputs; interaction overview diagrams describe more complete interactions, and thus more general, and system-wide, properties.
- Clocks (and time diagrams) are used to add the time dimension to systems, constrain the behavior of components, and be able to predicate on it.

All these diagrams supply users with a complete, homogeneous set of concepts to render the system-to-be in a consistent way, offer a “simple” coherent notation and keep the verification phase simple.

The rest of paper is organized as follows. Section 2 provides an overview of the MADES approach. Section 3 presents the background concepts of the semantic notation. Section 4 defines the *verification* notation, while Section 5 uses an example system to introduce the formal semantics. Section 6 surveys some related approaches and Section 7 concludes the paper.

2 Modeling and Verification Workflow

The MADES workflow is designed to allow users to carry out formal verification activities while hiding from them the details of the creation of the formal model and also of the execution of the verification phase itself. Two issues are key in this approach: (i) the notation used for modeling the system to be verified must be one with which the user is familiar with (ii) the verification phase must be carried out without user intervention, in a “push-button” manner.

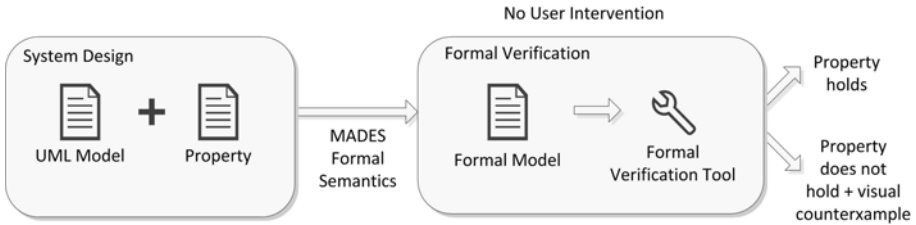


Fig. 1. Overview of the MADES workflow

Figure 1 provides an overview of the workflow. It starts with the definition by the user of a UML model of the system to be verified. By design, in the MADES approach the modeling notation conforms to the UML standard (including a pair of relevant profiles such as SysML [16] and MARTE [14]), with some restrictions that are needed to make the models verifiable in a fully automated way. In addition, the user provides as input the property to check the model against. We will discuss later how this property can be expressed.

Both the UML model and the property to be checked are translated automatically, without user intervention, into a formal model that is suitable to be input to a push-button formal verification tool. The translation is performed using the semantics described in Section 5. Then, the tool is run on the system/property combination, and its result is output. This can be either the notification that the property holds or, if the property does not hold, a trace of the system that violates it. In the second case, the counterexample trace is shown in UML form as feedback. The trace can be used to examine the behavior of the objective system.

Figure 2 shows a more detailed view of the MADES workflow, which highlights a distinguishing feature of the MADES approach, namely its inclusiveness with respect to the set of UML diagrams taken into account. In fact, unlike most existing approaches (see also Section 6), MADES allows users to use and combine a rich variety of UML diagrams to define the behavior of the system being designed. More precisely, the set of UML diagrams taken into consideration consists of class diagrams, object diagrams, sequence diagrams (SD), interaction overview diagrams (IOD), and state diagrams. Class diagrams and object diagrams provide an high-level overview of: (i) the types of the components of the system, (ii) the instances (i.e., the objects) of such types that are actually

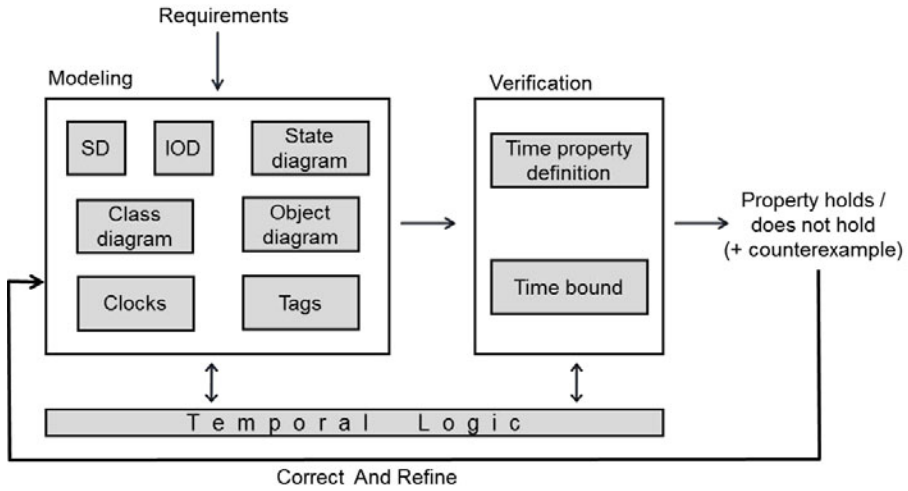


Fig. 2. Detailed workflow, with indication of available UML diagrams

present in the system, and (iii) their interconnections. Essentially, these diagrams provide the *alphabet* of the formal model, i.e., the basic items and events whose dynamics are described through the behavioral diagrams. State diagrams are used to describe the behavior of the components of the system taken one by one, in terms of their operations and of the effect that these have on their states (i.e., the attributes). Sequence diagrams, instead, define the basic interactions between the components introduced through class and object diagrams; these basic interactions are in turn composed into more complex ones through interaction overview diagrams. In addition, MARTE clocks can be used as reference to express timing constraints between events in the aforementioned diagrams. Finally, MADES allows users to add to diagrams some domain-specific tags that can be used to optimize the verification phase (these tags will not be discussed in this paper; an overview of them can be found in [1]).

Section 4 described the meta-model of the UML subset that designers can use to model systems to be verified in the MADES approach. To facilitate formal verification of MADES models, this subset of UML must be given a semantics based on a formalism that is at the same time *flexible*, to capture the meaning of heterogeneous diagrams, and *decidable*, to allow for fully automated verification. In addition, the formalism must be able to express the timing constraints described in the MADES notation through clocks. Given such requirements, the underlying formalism used in the MADES approach is the TRIO metric temporal logic described in Section 3, and in particular the decidable subset thereof that is supported by the Zot bounded model/satisfiability checker (hence the need to specify a time bound for the verification phase, as shown in Figure 2).

Finally, let us point out that, in the MADES approach, the property to be verified for the system can be expressed in two different ways. The first one is through the underlying formalism which is used to formalize the system (in our case, TRIO); this, however, would be against the idea that the user should never see the underlying formal notation. The second one is through a UML-based graphical notation that expresses the property of interest and that can be translated into the underlying formalism. The set of properties that can be expressed in TRIO is bigger than the set of properties that can be expressed through the UML-like notation. However UML hides the complexity of the TRIO language to the user. The UML-like graphical notation is still on-going research, thus this aspect of the work flow will not be discussed in this paper; some considerations in this regard can be found in [1][2].

3 TRIO and Zot

TRIO [7] is a first-order linear temporal logic that supports a metric on time. TRIO formulae are built out of the usual first-order connectives, operators, and quantifiers, as well as a single basic modal operator, called *Dist*, that relates the *current time*, which is left implicit in the formula, to another time instant: given a time-dependent formula F (i.e., a term representing a mapping from the time domain to truth values) and a (arithmetic) term t indicating a time distance (either positive or negative), the formula $\text{Dist}(F, t)$ specifies that F holds at a time instant whose distance is exactly t time units from the current instant. $\text{Dist}(F, t)$ is in turn also a time-dependent formula, as its truth value can be evaluated for any current time instant, so that temporal formulae can be nested as usual. While TRIO can exploit both discrete and dense sets as time domains, in this paper we assume the standard model of the nonnegative integers \mathbb{N} as discrete time domain. For convenience in the writing of specification formulae, TRIO defines a number of *derived* temporal operators from the basic *Dist*, through propositional composition and first-order logic quantification. Table 1 defines some of the most significant ones, including those used in this paper.

The TRIO specification of a system consists of a set of basic *items*, which are primitive elements, such as predicates, time-dependent values, and functions, representing the elementary phenomena of the system. The behavior of a system over time is described by a set of TRIO formulae, which state how the items are constrained and how they vary, in a purely descriptive (or declarative) fashion.

The goal of the verification phase is to ensure that the system S satisfies some desired property R , that is, that $S \models R$. In the TRIO approach S and R are both expressed as logic formulae Σ and ρ , respectively; then, showing that $S \models R$ amounts to proving that $\Sigma \Rightarrow \rho$ is valid.

TRIO is supported by a variety of verification techniques implemented in prototype tools. In this paper we use *Zot* [17], a bounded satisfiability checker which supports verification of discrete-time TRIO models. *Zot*³ encodes satisfiability

³ <http://home.dei.polimi.it/pradella/Zot>

Table 1. TRIO derived temporal operators

OPERATOR	DEFINITION
Past(F, t)	$t \geq 0 \wedge \text{Dist}(F, -t)$
Futr(F, t)	$t \geq 0 \wedge \text{Dist}(F, t)$
Alw(F)	$\forall d : \text{Dist}(F, d)$
AlwP(F)	$\forall d > 0 : \text{Past}(F, d)$
AlwF(F)	$\forall d > 0 : \text{Futr}(F, d)$
SomF(F)	$\exists d > 0 : \text{Futr}(F, d)$
SomP(F)	$\exists d > 0 : \text{Past}(F, d)$
Lasted(F, t)	$\forall d \in (0, t] : \text{Past}(F, d)$
Lasts(F, t)	$\forall d \in (0, t] : \text{Futr}(F, d)$
WithinP(F, t)	$\exists d \in (0, t] : \text{Past}(F, d)$
WithinF(F, t)	$\exists d \in (0, t] : \text{Futr}(F, d)$
Since(F, G)	$\exists d > 0 : \text{Lasted}(F, d) \wedge \text{Past}(G, d)$
Until(F, G)	$\exists d > 0 : \text{Lasts}(F, d) \wedge \text{Futr}(G, d)$

(and validity) problems for discrete-time TRIO formulae as propositional satisfiability (SAT) problems, which are then checked with off-the-shelf SAT solvers. More recently, we developed a more efficient encoding that exploits the features of Satisfiability Modulo Theories (SMT) solvers [3]. Through *Zot* one can verify whether stated properties hold for the modeled system (or parts thereof) or not; if a property does not hold, *Zot* produces a counterexample that violates it.

4 A Verifiable Subset of UML

This section presents, through a set of UML class diagrams, the meta-model of the verification notation. The meta-model is divided into the packages as shown in Figure 3.

The fundamental elements of the notation are grouped together in the *core* package, which includes different class diagrams. *core.diagrams* (shown in Figure 4) describes the set of diagrams used in the verification workflow and how they are related to one another. The MADES model is composed by Class diagrams, Object diagrams, and Interaction Overview Diagrams (IOD). State diagrams describe the behavior of the objects belonging to a certain class, and Sequence diagrams show the details of the interactions between objects.

Diagram *core.types* (Figure 5) gives an overview of the data types that can be used in the verification workflow. A *TypedElement* is an element of the system that has a type. A type can be one of the classes declared in the class diagram, or a *DataType*. A *DataType* can be a *PrimitiveType*, an *Array* or an *Enumeration*. Primitive types are *Boolean*, or *Integer*. An *Array* is an ordered list (of fixed size) of *Integers*. An *Enumeration* is a finite set of *Integers*.

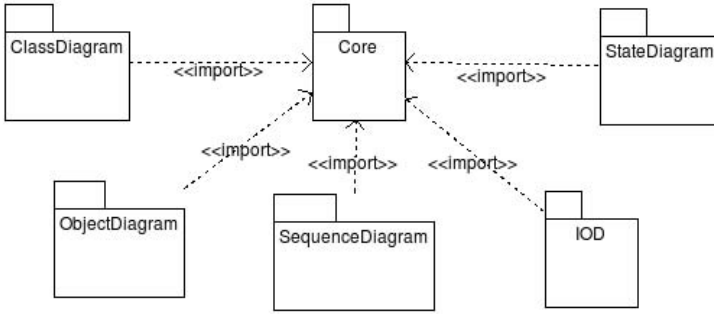


Fig. 3. Metamodel packages

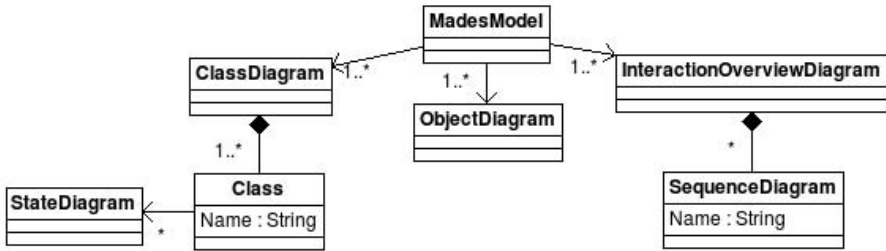


Fig. 4. MADES metamodel: core.diagrams

Diagram *core.events* (shown in Figure 6) defines what is an *Event*. Events are directly translated into temporal logic predicates and define how the system proceeds over time. Their temporal relationships will be precisely defined in Section 5; here, we simply list them with their informal meaning. *ActivityStart* and *ActivityEnd* occur in the time instants in which the IOD activity starts and ends. *DecisionPath* and *ForkPath* correspond to the time instants in which the IOD takes a certain path after a decision/fork operator. *JoinEnd* is the time instant in which all the diagrams preceding a certain IOD Join operator complete their execution. *SDStart* and *SDEnd* occur when a certain sequence diagram starts/ends. *SDAct* is the time instant in which a certain sequence diagram is ready to start. *SDAct* immediately precedes *SDStart*. *MessageStart* and *MessageEnd* occur when a certain message starts/ends. *ExOccStart* and *ExOccEnd* occur when a certain execution occurrence starts/ends. A *TimeEvent* is a *TimedInstantObservation* in a certain sequence diagram (for details see UML::CommonBehaviours::SimpleTime from [16]). The time note @t1 in Figure 12 of Section 5 shows an example of *TimeEvent*. Finally, *Interrupt* is the time instant in which a certain interrupt (i.e., an event that causes activities in an interruptible region of a IOD to exit, see [16] for further details) occurs. The rest of the metamodel describes how these events are associated to elements in the various behavioral diagrams, as shown, for example, in Figure 8.

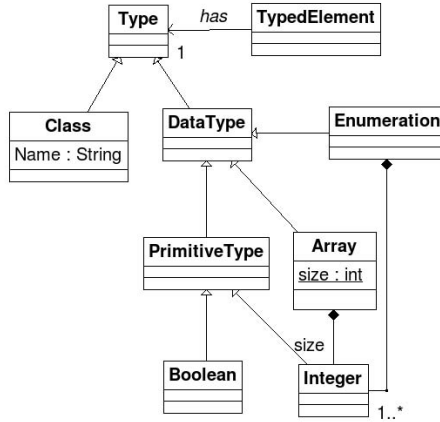


Fig. 5. MADES metamodel: core.types

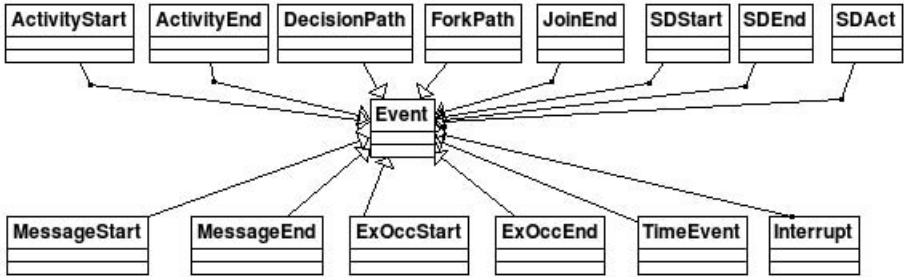


Fig. 6. MADES metamodel: core.events

Diagram *core.clocks* (Figure 7) defines the features of clocks. With respect to the UML/MARTE notion of clocks, for formal verification purposes we deal only with discrete clocks. Clock types are defined in the class diagram of a MADES model. Class *ClockType* has a set of attributes that define specific features of the clocks of that type (e.g., their period). A *Clock* has a *ClockType* and it can be attached to objects, classes, and sequence diagrams. When a clock is attached to an object (resp. class) the intuitive semantics is that the events related to that object (resp. to the objects belonging to the class) will proceed with the tick of this clock. When a clock is attached to a sequence diagram, all the events of the objects inside the sequence diagram will proceed with the tick of this clock (discrepancies between, for example, the clock of an object and that of a sequence diagram in which that object appears can be highlighted and sorted out during the verification phase).

The *core* package is completed with the *core.expressions* diagram (not shown here for the sake of brevity), which defines what is a valid expression. Intuitively,

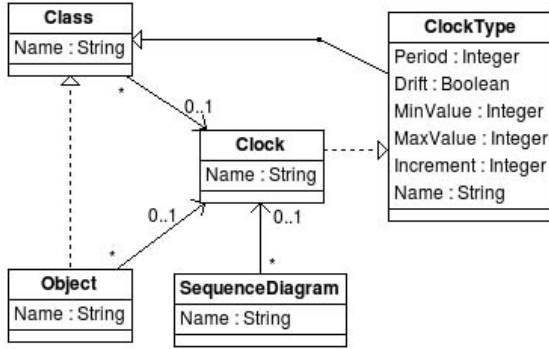


Fig. 7. MADES metamodel: core.clocks

the MADES verification admits three types of expressions: *MathematicalExpression*, *BooleanExpression*, *TimeExpression*. These expressions can be used in *Assignments*, whose meaning is intuitive. More precisely, *MathematicalExpressions* can be used in *Assignments* to variables of Integer type. *BooleanExpressions* can be used in *Assignment* to variables of Boolean type, and everywhere a boolean value is admitted (for example in the decision operator of the IOD). *TimeExpressions* can be used to define time constraints between events.

The other packages of the MADES metamodel describe the single diagrams in details. The MADES verification notation does not impose restrictions on the operators of Class diagrams, nor on those of Object diagrams, for which we refer to the UML specification [16]. As shown in Figure 8, a *SequenceDiagram* can contain: *Messages*, *ExecutionOccurrences*, *TimeEvents*, *CombinedFragments* and *StateInvariants*. *RecursiveMessages* are a special type of messages which are used not only for self invocations, but also for assignments to variables of *Integer* and *Boolean* type. Finally, *TimeConstraints* can be attached to sequence diagrams to define relations between the time events defined in the diagram. A *TimeConstraint* is a boolean expression made of *TimeInequalities* that relates two different events with some inequality operator. If the sequence diagram is inside an interruptible region of the IOD of the system, then *TimeConstraints* may also refer to those interrupts (as an example see Figure 12). The diagram of Figure 8 summarizes those concepts and shows the relations between the operators of sequence diagram and the events associated to them. Those events will be translated into temporal logic predicates together with the axioms that define their precise semantics.

InteractionOverviewDiagrams (whose class diagram is not shown here for the sake of brevity) can be seen as activity diagrams whose nodes are sequence diagrams. This very simple definition hides a number of details that are needed in order to define a precise semantics. In particular, in the MADES metamodel the *SequenceDiagrams* that are part of an *InteractionOverviewDiagram* have

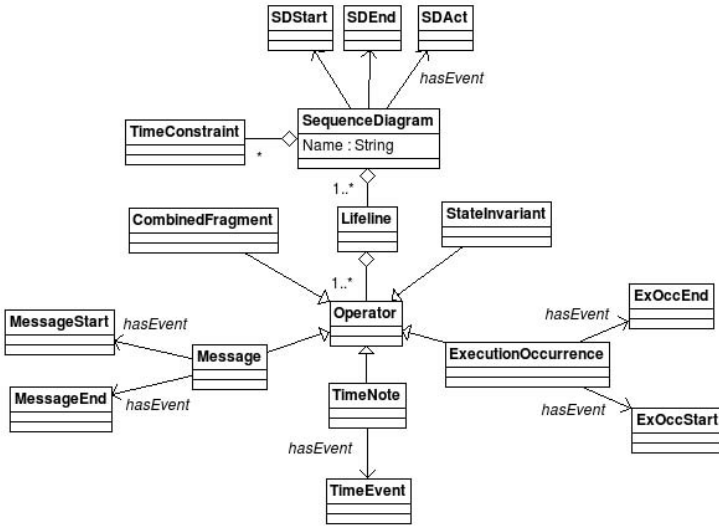


Fig. 8. MADES metamodel: The main elements of sequence diagrams

one incoming and one outgoing flow, and they can be grouped together through *InterruptibleRegions*. The *ExceptionEdge* going out from the *InterruptibleRegion* defines the name of the *Interrupt* associated with that region. This interrupt can be caught by different *InterruptNodes* that go into the *SequenceDiagram* which is responsible to continue the execution after that event. Figure 11 shows an example of *InterruptibleRegion*, which is associated with interrupt *connTimeout* that is caught by the *reconnect* sequence diagram through a suitable *InterruptNode*.

Finally, regarding *StateDiagrams* (whose metamodel is not shown here) we decided to keep the specification as simple as possible. In particular, a *StateDiagram* is a set of *States*. *InitialState* and *FinalState* are two particular kinds of *States*. The *Transition* from one state to another can be triggered by the *Events* of the system (i.e., those defined in Figure 6). Also, a *Transition* can be taken only if its *Guard* is true. The *Guard* is a *BooleanExpression* (whose features are described above). An example of state diagram is shown in Figure 13.

5 From UML to Temporal Logic Formal Semantics

This section presents the MADES semantics through an example system, which includes some desired properties to be verified. The system that is used to illustrate the modeling and verification features of the MADES approach is a simple telephone system. After a brief description of the system, this section shows how some meaningful UML diagrams are translated into their corresponding temporal logic form according to the MADES semantics.

5.1 Telephone System

The telephone system should provide the following features: At startup the system should connect to the remote server and initialize the graphical user interface (GUI). If the telephone is not correctly connected to the server, the GUI will not be shown. The connection is attempted 3 times with a timeout of 10 seconds. When the startup is finished, the system is ready to receive incoming calls and SMSs. Incoming calls may arrive at any instant. Incoming SMSs are checked on the server every 20 seconds by the telephone itself. If no reply is received by the server within 10 seconds, the attempt is not repeated. If the download is not completed within 10 seconds, the download is repeated. If the telephone is idle (e.g. it is not performing any call, nor an SMS composition) and the user presses a number, the number itself is shown on the screen and the telephone waits for the remaining digits until the green button is pressed. If the red button is pressed the system aborts the operation. If the SMS is not sent within 15 seconds, the operation is aborted. If the telephone is idle and the user presses the ok button, then a textual interface is shown to compose the SMS. When the ok button is pressed again the GUI waits for the telephone number and when the ok button is pressed again the SMS is sent to the recipient. SMSs are sent with tokens of 160 characters. The transmission time follows this formula: $trTime = length(SMS)/sigStrength * 10sec$, where $sigStrength$ may be [1..5].

5.2 UML Diagrams and Their Formal Semantics

Figure 9 shows the class diagram of the system. The diagram is not itself translated into temporal logic, though it is used to determine the *alphabet* of the formal model, that is, the actual predicates that appear in the formal model; for example, from the names of the operations the corresponding events are defined. The class diagram also contains some MADES-specific tags (e.g. the «TI» stereotype) that will not be analyzed in this paper. The diagram is also used to define the clock types of the system. For example, class *SMSClockType* defines a type of clock whose period is 20, which will be used for the periodic SMS retrieval. Figure 10 shows the object diagram of the telephone system, which contains the objects that are taken into consideration during the actual verification phase. The number of objects (i.e., instances of classes described in the class diagram) considered in the verification model must be finite, to allow for full automated verification. The role of the object diagram, then, is to precisely define what instances are present in the system model, and their (finite) numbers. For example, the diagram of Figure 10 defines that there are six *TransmissionThreads*. These objects are tagged as being a «set». The semantics that is given in the MADES approach to this stereotype is that, when an operation is invoked on one of the objects of a set, the actual identity of the object is irrelevant, as the objects all behave in the same manner. In the future, we will use such information to optimize the verification phase, but we do not delve into this issue any further in this paper. Figure 10 also shows an instance of clock *SMSClockType*.

The Interaction Overview Diagram of Figure 11 shows the startup of the system. Sequence diagrams are used to group together macro-operations which

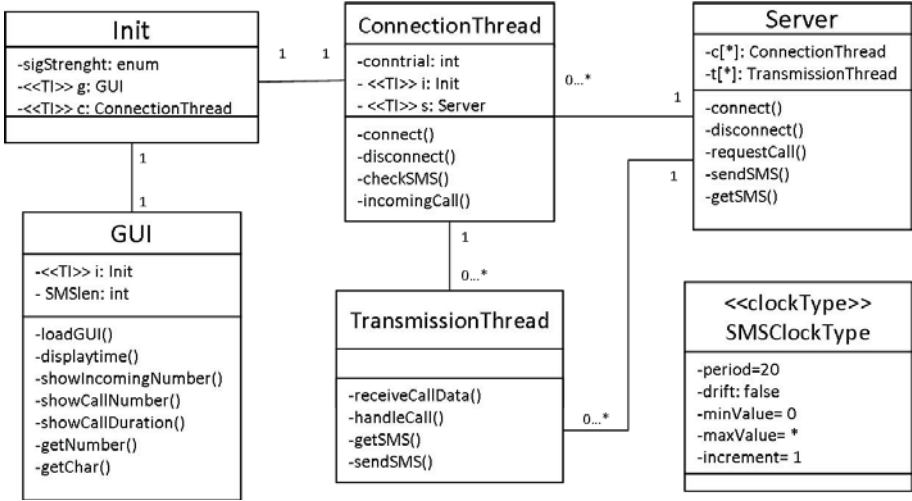


Fig. 9. Telephone System Class Diagram

are then combined together to obtain the complete system specification. Interruptible regions are used to stop the behaviors occurring inside a certain set of sequence diagrams and continue elsewhere. In this particular example we model the fact that while the telephone is performing the connection to the server it may happen that a connection timeout occurs. In that case the *connect* sequence diagram is stopped, and the *reconnect* sequence diagram continues the execution.

The temporal logic semantics of the diagram is generated as described in the following. Each sequence diagram D_x has three events, namely D_xAct , D_xStart , D_xEnd . Each event is translated into one temporal logic predicate, thus predicate D_xAct holds when the diagram is ready to start its execution, predicate D_xStart holds exactly one time unit later, and predicate D_xEnd holds when the diagram terminates. Depending on the operator that precedes diagram D_x the formula that defines D_xAct may change; for reasons of brevity, a presentation of the complete algorithm that manages all different cases is outside the scope of this paper (we refer the interested reader to [2]). In the following we focus on the definitions concerning some of the elements of Figure 11.

When the whole diagram starts (i.e., predicate *ActivityStart* holds), the first sequence diagram (i.e., *init*) is activated. In addition, diagram *loadGUI* is activated when *init* ends, as represented by the fork operator between the two diagrams. These properties are formalized by formulae (1)-(2).

$$InitAct \Leftrightarrow ActivityStart \tag{1}$$

$$loadGUIAct \Leftrightarrow InitEnd \tag{2}$$

$$connectAct \Leftrightarrow InitEnd \vee reconnectEnd \tag{3}$$

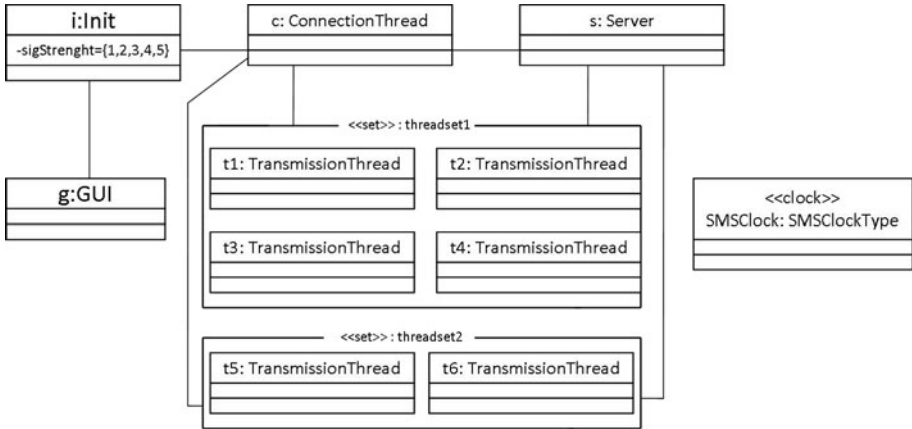


Fig. 10. Telephone System Object Diagram

If we focus on the *connect* sequence diagram of Figure 11, we notice that its activation condition holds at the same time instant in which either diagram *init* ends, or diagram *reconnect* ends. The reason is that *connect* is preceded by a merge operator, thus both paths entering the latter may activate diagram *connect*. One of those paths originates from a fork operator, but this is *transparent* to the semantics. This is all formalized by formula (3).

If, on the other hand, we analyze diagram *run*, its activation condition holds exactly when the nodes preceding the join operator have finished their execution. Formula (5) defines when the join ends (i.e., when the last between *connect* and *loadGui* ends). Formula (4) states that diagram *run* is ready to start exactly in the same time instant in which the join ends.

$$runAct \Leftrightarrow Join1End \quad (4)$$

$$Join1End \Leftrightarrow (loadGUIEnd \wedge Since(\neg Join1End, connectEnd)) \vee (connectEnd \wedge Since(\neg Join1End, loadGUIEnd)) \quad (5)$$

To conclude this part of the semantics, formulae (6)-(7) define the relations between the activation event and the start event of sequence diagram *connect*. More precisely, $D_x Act$ holds when the enabling conditions are true. Then, if the diagram is enabled and in the next time instant the activity has not ended, $D_x Start$ holds in the next time instant. Similar formulae hold for all other diagrams in the figure (they can be obtained simply by replacing *connect* with the name of the other diagrams, e.g., *loadGUI*).

$$connectAct \wedge \neg Futr(ActivityEnd, 1) \Rightarrow Futr(connectStart, 1) \quad (6)$$

$$connectStart \Rightarrow Past(connectAct, 1) \quad (7)$$

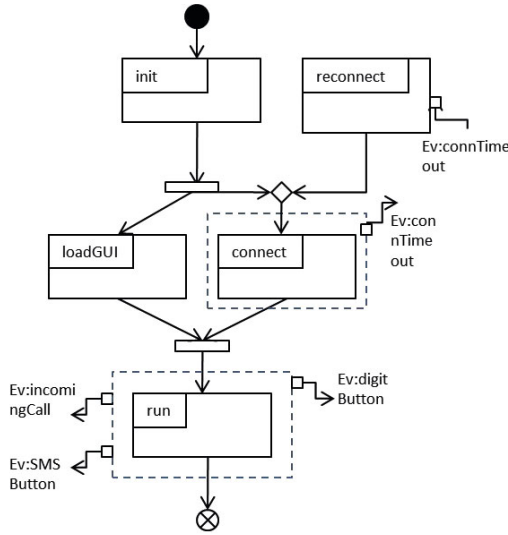


Fig. 11. Telephone System Interaction Overview Diagram

Figure 12 shows the *connect* sequence diagram in detail. The instance *c* of *ConnectionThread* calls its own procedure *connect()* to start the connection to the server. The time instant in which this connection procedure starts is marked with the time note @*t1*. Inside the *connect()* procedure the object *c* invokes the *connect()* procedure of the instance *s* of *Server*. After a while the reply message is received and the sequence diagram ends. According to the system specification the connection is attempted 3 times with a timeout of 10 seconds. To model the connection timeout of 10 seconds we added a *time constraint* to the sequence diagram. A time constraint relates two events with some temporal inequality operator. In that case the semantics is that if the timeout occurs, then it must occur exactly 10 time units after @*t1* (more precisely, the constraint says that the difference between the time of the timeout and the time of event @*t1* is exactly 10). Finally, the *alternative combined fragment* is added to specify that the connection is attempted only if variable *conntrial* (which is an attribute of class *ConnectionThread*) is strictly less than 3.

The temporal logic semantics starts from the events that are extracted from the diagram. In this case we have the following events: *connectSDStart*, *connectSDEnd*, which correspond, respectively, to the start and end of the whole diagram; *t1Event*, which represents the time instant in which event @*t1* occurs; *c.connect1Start*, *c.connect1End*, which correspond to the start and end of recursive message *connect* (i.e., the invocation of operation *connect* being made by *c* on itself); *c.exOcc1Start*, *c.exOcc1End*, which represent the start and end of the execution occurrence that covers the lifeline of instance *c*; *s.connect1Start*, *s.connect1End*, which correspond, respectively, to message (i.e., invocation)

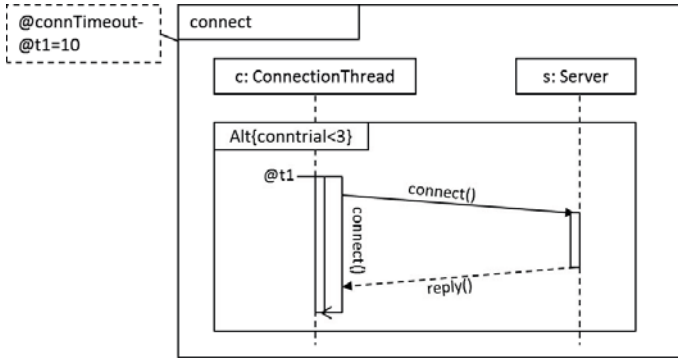


Fig. 12. Telephone System SD connect

connect being sent by object *c* to object *s*, and to the message being received by object *s*; *s.connect1ReplyStart*, *s.connect1ReplyEnd*, which correspond, respectively, to message *reply* (i.e., the reply to the invocation of operation *connect*) being sent by object *s*, and to the message being received by object *c*; *s.ExOcc1Start*, *s.ExOcc1End*, which represent the start and end of the execution occurrence that covers the lifeline of instance *s*. Notice that the events related to the execution occurrences and to the messages are labeled with some index. This is due to the fact that each object can have more than one execution occurrence in the system, and the temporal logic predicates must distinguish between them. The same holds for each method invocation. The events which are placed graphically on the same y-axis of the same lifeline hold on the same time instant, no matter what the other axioms state. This is defined by formulae (8)-(11).

$$t1Event \Leftrightarrow c.connect1Start \wedge c.ExOcc1Start \quad (8)$$

$$s.connect1End \Leftrightarrow s.ExOcc1Start \quad (9)$$

$$s.connect1ReplyStart \Leftrightarrow s.ExOcc1End \quad (10)$$

$$c.connect1End \Leftrightarrow c.ExOcc1End \quad (11)$$

Formulae (12)-(13), instead, enforce the ordering between the different events of the diagram. In particular, the fact that an event Ev_i is followed by another event Ev_j is stated by formula (12). On the other hand, formula (13) entails that we must have an occurrence of Ev_i in the past in order to have Ev_j now. In addition formula (12) defines that, if Ev_i holds now, then we must consider the following possibilities: either it exists in the future a time instant in which Ev_j holds and for all the time instants between Ev_i and Ev_j the sequence diagram is not interrupted, or it exist in the future a time instant in which the diagram is interrupted, and until that time Ev_j does not occur. The SD can be interrupted for two reasons: either because the interaction overview diagram

ends, thus *ActivityEnd* holds, or because an interrupt associated to that diagram occurs. Formula (I3) states similar properties for the past.

$$Ev_i \Rightarrow \text{Until}(\neg Ev_i \wedge \neg \text{ActivityEnd} \wedge \neg \text{Interrupt}_i \dots \wedge \neg \text{Interrupt}_k, Ev_j) \vee \text{Until}(\neg Ev_i \wedge \neg Ev_j, \text{ActivityEnd} \vee \text{Interrupt}_i \dots \vee \text{Interrupt}_k) \tag{12}$$

$$Ev_j \Rightarrow \text{Since}(\neg Ev_j \wedge \neg \text{Interrupt}_i \dots \wedge \neg \text{Interrupt}_k \wedge \neg \text{ActivityEnd}, Ev_i) \tag{13}$$

Considering the diagram of Figure I2 formulae (I2)-(I3) are instantiated with the following events:

<i>Ev_i</i>	<i>Ev_j</i>
c.connect1Start	s.connect1Start
s.connect1Start	s.connect1End
s.connect1Start	s.connect1ReplyEnd
s.connect1End	s.connect1ReplyStart
s.connect1ReplyEnd	c.connect1End
c.connect1End	connectSDEnd

Moreover in this case we have that the set *Interrupt_i...Interrupt_k* is reduced to *connTimeout*.

Because the diagram includes an *Alternative combined fragment* the ordering between *connectSDStart* and the first event of the combined fragment is treated separately. Namely, we specify that *t1Event* follows *connectSDStart* only if *conntrial* is less than 3. This means that *conntrial < 3* is added to the precondition of formula (I2) instantiated with *Ev_i = connectSDStart* and *Ev_j = c.connect1Start*. Finally, we instantiate again formula (I2) with *Ev_i = connectSDStart* and *Ev_j = connectSDEnd* with *conntrial ≥ 3* included in the sufficient condition as before.

The time constraint attached to the diagram is translated into formula (I4) which states that *connTimeout* occurs exactly 10 time units after *t1Event*. Notice that according to formula (8) this is the same time instant in which the *connect* self-recursive message starts. Also notice that if the *connTimeout* occurs when the diagram is finished this does not affect the normal behavior of the sequence diagram according to formulae (I2)-(I3).

$$\text{connTimeout} \Leftrightarrow \text{Past}(t1Event, 10) \tag{14}$$

State diagrams complete the picture of what is taken into consideration by the MADES formal verification notation. The *ConnectionThread* state diagram is composed of five states (see Figure I3). The initial state goes into the *start* state. Here the object waits the beginning of the connection procedure. The self-message *c.connect()* moves it from *start* to *connecting*. The *s.connect()* procedure call moves it to the *Waiting* state. If the *connTimeout* interrupt occurs it goes to *reconnect* and then to *connecting* again with *c.connect()*. If *connTimeout* does not occur, the *c.connectEnd* event moves the object into

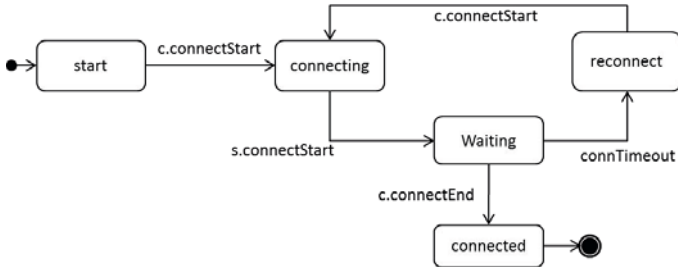


Fig. 13. Telephone System ConnectionThread State Diagram

the *connected* state. Let us focus on the definition of the behavior that makes the state machine enter and exit the *connecting* state. Predicate $c_Connecting$ represents when the diagram is in state *connecting*. Then, formulae (15)-(16) state, essentially, that $s_connectStart$ is the only event responsible for going out of the *connecting* state. Formula (17) instead, enforces the fact that the states are mutually exclusive. The same kind of formulae hold for the other states of the state diagram. Additional formulae, not shown here for the sake of brevity, define that in the first time instant the machine is in the initial state, then it non-deterministically enters the *start* state because no event is defined in the transition label.

$$c_Connecting \wedge Futr(c_Waiting, 1) \Rightarrow s_connectStart \quad (15)$$

$$c_Connecting \wedge Futr(c_Connecting, 1) \Rightarrow \neg s_connectStart \quad (16)$$

$$c_Connecting \Rightarrow \neg c_Start \wedge \neg c_Waiting \wedge \neg c_Reconnect \wedge \neg c_Connected \quad (17)$$

Finally, the complete formal model Σ that arises from the diagrams describing the system is given by the simple logic conjunction of all formulae. The result is a model that includes all features and constraints defined by the various diagrams, which allows us to formally perform various checks through tools such as the *Zot* bounded model/satisfiability checker.

A first kind of verification that can be carried out is a check of the consistency of the model obtained by the combination of the various diagrams. This corresponds to determining whether there exists at least an execution that is compatible with the system model, a task that can be carried out through the *Zot* tool simply by checking whether formula Σ is satisfiable or not.

As depicted in Figure 2, the MADES approach also allows users to define temporal properties of interest of the system, to be checked through formal verification techniques. For example, in the case of the telephone system presented in this section, a property we may be interested in is the following:

”The telephone start-up lasts less than 30 seconds”

To express this property formally in a graphical, user-friendly, way, we are currently working on a visual notation that is inspired by sequence diagrams. The

definition of the precise syntax and semantics of such notation is left for future work. Here, we simply note that, since the semantics of the MADES verification notation is given in terms of a formal language, properties of interest can always be expressed using this underlying formalism, in our case the TRIO metric temporal logic. In the case of the temporal property mentioned above, its formalization in the TRIO language could be the following

$$\mathit{initSDStart} \Rightarrow \mathit{WithinF}(\mathit{runSDStart}, 30) \quad (18)$$

where $\mathit{initSDStart}$ and $\mathit{runSDStart}$ are, respectively, the events associated with the beginning of the init and run sequence diagrams.

A wide range of temporal properties, of which (18) is just a simple example, can be defined by using the full set of features of the TRIO temporal logic to predicate on the events and attributes of the system modeled through the MADES notation. The properties that users are interested in verifying, however, are often a small subset of those that can be expressed through the full power of the TRIO language. For this reason, the graphical notation we are developing within the MADES project to express properties to be verified will trade the full expressive power of the TRIO language for a higher degree of simplicity and intuitiveness of the representation of properties of interest.

6 Related Work

The vast majority of works that ascribe UML with a formal semantics usually concentrate on individual diagrams. Only few approaches tried to give a semantics that addresses different diagrams. For example, Hansen et al. [13] describe a translation of a subset of executable UML (xUML) into the process algebraic specification language mCRL2. The subset consists of class diagrams, state machines, and an action language that complies with the UML action semantics. This approach does not take into account sequence diagrams and it only concentrates on well-defined fairness and safety properties.

Diethers and Huhn [9] present Voodoo, a tool to automatically verify whether a set of statechart diagrams that model a system satisfies communication and timing constraints given as sequence diagrams. Both types of diagrams are translated into timed automata for the verification. Also Damm et al. [8] define the semantical foundation of a sublanguage of UML that is mostly based on statechart diagrams. The semantics is given in terms of symbolic transition systems, and it mostly addresses the concurrency and communication between objects.

Burmester et al. [5] exploit real-time component diagrams and real-time statechart diagrams to model the static and dynamic parts of a system. These diagrams are formalized in terms of hierarchical timed automata, which allow the authors to run compositional verifications of partial models. Saldhana and Shatz [19] describe a methodology to develop a Petri net of the system. They derive an Object Petri Net Model (OPM) from statechart diagrams connected through collaboration diagrams. The analysis is carried out by exploiting the usual techniques for Petri nets.

As mentioned above there are many more works that focus on the separate formalization of single diagrams. Hammal [12] defines a method for translating statechart diagrams into Interval Timed Petri Nets (ITPN) to run consistency analyses. The ITPN enables the analysis of performance and time properties of complex systems. Störrle [20] investigates the alignment activity diagrams to Petri nets. It provides a mapping of the basic elements of activity diagrams onto Petri nets and discusses the problems that arise from this translation. Es-huis [10] proposes two translations from activity diagrams to the input language of NuSMV, a well-known symbolic model checker. Both translations map activity diagrams into finite state machines and are inspired by existing semantics for statechart diagrams. Finally, Cengarle and Knapp [6] investigate interaction diagrams and provide an operational semantics for them, while Tebibel [4] uses hierarchical colored Petri nets to define a formal semantics for interaction overview diagrams.

7 Conclusions

This paper builds on the need for formal UML dialects for the design and validation of timed systems. It starts from the MADES modeling notation, which is a particular extension to UML that borrows concepts from SysML and MARTE, and it proposes a formal semantics for a verification-oriented version of the language. The verification notation can be seen as an “abstract” notation for the complete design notation. It filters out irrelevant elements and ascribes a formal semantics to the other ones. The formal semantics covers a wide range of UML diagrams and concentrates on time-related aspects. The formal semantics and the verification tools will be used in the MADES project for the early verification of embedded systems.

The paper outlines the formal semantics for the verification notation based on the TRIO temporal logic. The definition is provided in a modular and methodological way to let the reader understand how the different pieces fit together. Finer-grained improvements, optimizations, and tailoring are part of our ongoing work. The complete verification tool suite implementing the approach described in this paper will be tested for usability by the industrial partners of the MADES project.

Acknowledgments. This research was supported by the European Community’s Seventh Framework Program (FP7/2007-2013) under grant agreement n. 248864 (MADES), and by the Programme IDEAS-ERC, Project 227977-SMScom.

References

1. Baresi, L., Morzenti, A., Motta, A., Rossi, M.: D3.1 domain-specific and user-centred verification. Technical report, MADES Consortium (2010)
2. Baresi, L., Morzenti, A., Motta, A., Rossi, M.: D3.3 formal dynamic semantics of the modelling notation. Technical report, MADES Consortium (2011)

3. Bersani, M.M., Frigeri, A., Pradella, M., Rossi, M., Morzenti, A., San Pietro, P.: Bounded reachability for temporal logic over constraint systems. In: Proc. of the Int. Symp. on Temporal Representation and Reasoning (TIME), pp. 43–50 (2010)
4. Bouabana-Tebibel, T.: Semantics of the interaction overview diagram. In: Proc. of the IEEE Int. Conf. on Information Reuse Integration (IRI), pp. 278–283 (2009)
5. Burmester, S., Giese, H., Hirsch, M., Schilling, D., Tichy, M.: The fujaba real-time tool suite: model-driven development of safety-critical, real-time systems. In: Proc. of the 27th Int. Conf. on Soft. Eng., ICSE 2005, pp. 670–671 (2005)
6. Cengarle, M.V., Knapp, A.: Operational semantics of UML 2.0 interactions. Technical Report TUM-I0505, Technische Universität Munchen (2005)
7. Ciapessoni, E., Coen-Portisini, A., Crivelli, E., Mandrioli, D., Mirandola, P., Morzenti, A.: From formal models to formally-based methods: an industrial experience. ACM TOSEM 8(1), 79–113 (1999)
8. Damm, W., Josko, B., Pnueli, A., Votintseva, A.: A discrete-time uml semantics for concurrency and communication in safety-critical applications. Sci. Comput. Program. 55, 81–115 (2005)
9. Diethers, K., Huhn, M.: Voodoo: Verification of object-oriented designs using uppaal. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 139–143. Springer, Heidelberg (2004)
10. Eshuis, R.: Symbolic model checking of UML activity diagrams. ACM Trans. Softw. Eng. Methodol. 15(1), 1–38 (2006)
11. Eshuis, R., Wieringa, R.: Tool support for verifying UML activity diagrams. IEEE Trans. Software Eng. 30(7), 437–447 (2004)
12. Hammal, Y.: A formal semantics of uml statecharts by means of timed petri nets. In: Wang, F. (ed.) FORTE 2005. LNCS, vol. 3731, pp. 38–52. Springer, Heidelberg (2005)
13. Hansen, H., Ketema, J., Luttkik, B., Mousavi, M., van de Pol, J.: Towards model checking executable uml specifications in mcrl2. Innovations in Systems and Software Engineering 6, 83–90 (2010)
14. Object Management Group. UML Profile for Modeling and Analysis of Real-Time Embedded Systems. Technical report, OMG, formal/2009-11-02 (2009)
15. Object Management Group. OMG Systems Modeling Language (OMG SysML). Technical report, OMG, formal/2010-06-01 (2010)
16. Object Management Group. OMG Unified Modeling Language (OMG UML), Superstructure. Technical report, OMG, formal/2010-05-05 (2010)
17. Pradella, M., Morzenti, A., San Pietro, P.: The symmetry of the past and of the future: bi-infinite time in the verification of temporal properties. In: Proceedings of ESEC/SIGSOFT FSE, pp. 312–320 (2007)
18. Pradella, M., Rossi, M., Mandrioli, D.: ArchiTRIO: A UML-compatible language for architectural description and its formal semantics. In: Wang, F. (ed.) FORTE 2005. LNCS, vol. 3731, pp. 381–395. Springer, Heidelberg (2005)
19. Saldhana, J.A., Shatz, S.M.: Uml diagrams to object petri net models: An approach for modeling and analysis. In: Proc. of SEKE 2000, pp. 103–110 (2000)
20. Störrle, H., Hausmann, J.H.: Towards a formal semantics of UML 2.0 activities. In: Software Engineering. Lec. Not. in Inf, vol. 64, pp. 117–128 (2005)

Generic Fault Modelling for Fault Injection

Rickard Svenningsson¹, Henrik Eriksson¹, Jonny Vinter¹, and Martin Törngren²

¹Department of Electronics, SP Technical Research Institute of Sweden
{rickard.svenningsson, henrik.eriksson, jonny.vinter}@sp.se

²Department of Mechatronics, KTH Royal Institute of Technology
martin@md.kth.se

Abstract. Fault injection is a widely used experimental dependability validation method, with a vast amount of techniques and tools. Within the scope of MOGENTES, an EU 7th framework programme project, tools have been developed which implements three different fault injection techniques; hardware-implemented fault injection, software-implemented fault injection and model-implemented fault injection. To support fault injection under the same conditions with these tools, an unambiguous description format for failure modes has been developed. Within MOGENTES, over 30 different failure modes have been identified, which all are implementable using the proposed format. XML has been chosen as the storage format for the failure modes, following a schema that is described in this paper.

1 Introduction

As fault injection (also known as fault insertion testing) has become widely used as an experimental dependability validation method, many different techniques for injecting faults have been developed. Fault injection accelerates the occurrences of faults in a system and the main purpose is to evaluate and debug error handling mechanisms. It is used at various abstraction levels and phases of the development process. Traditional fault injection is performed during the later part of the development process to verify that a system fulfills its robustness requirements, thus being a straightforward method where faults are injected during run-time to observe the system behavior under the influence of the injected faults. However, by using model-implemented fault injection [1] (MIFI), fault injection can be performed already into behavior models, i.e. during the early development phases. One of the major implications of using this technique is that the effects of faults cannot be automatically derived, since a system can be synthesized in virtually infinite ways. To cope with this, hardware faults are abstracted into high-level fault models that represent the fault effect on the same abstraction level as the behavior model, with the anticipated implementation in mind. In MOGENTES [2], more than 30 fault models have been created that represent possible effects of hardware faults. Performing fault injection on model level can be considered as an automated dynamic failure modes and effects analysis [3] (FMEA), and if exhaustively performed [4], it also covers the result of a dynamic fault tree analysis [5] (FTA). The process of choosing the fault models that shall be applied for every single fault injection location in the model is similar to the process of choosing failure modes for components in an FMEA, where failure modes are chosen from domain specific standards, e.g. the IEC TR 62380 reliability data handbook [6].

To create and store generic fault models in a tool-independent format, the authors propose an XML schema [7]. Many of the fault models defined in MOGENTES have been implemented in the proposed representation and successfully used in the tool MODIFI [4].

Similar work has been performed independently by others [8] to support interoperability between different fault injectors for SWIFI. The goal here is to standardize the information sent from a fault injection manager, where campaigns are created, to the fault injector, which injects faults on a specific target hardware. Applicability to three different fault injectors (Windows, i386, and Sparc) are demonstrated. Our goal extends this idea by standardizing the fault descriptions used for fault injection tools with fault injectors working at different levels: model, software and hardware, respectively.

The remainder of this paper is organized as follows: following this introduction is a section about model-implemented fault injection which demonstrates the use of high level fault models in the MODIFI tool. After that, a few fault models from the definitions in MOGENTES are presented, followed by the description of the proposed modeling format. Finally some concluding remarks are presented followed by references.

2 Model-Implemented Fault Injection in MODIFI

The MODIFI (MODEL-Implemented Fault Injection) tool injects faults in behavior models developed using the Matlab/Simulink [9] environment. The purpose of the tool is to carry out an early evaluation of model robustness against faults and to exercise and evaluate error handling mechanisms that are part of the model. Such mechanisms are sometimes impossible to evaluate without the usage of fault injection, e.g. when the default-statement in a C switch block is used to capture invalid values.

To illustrate how fault injection is performed in MODIFI, the model depicted in Figure 1 is used as a reference.

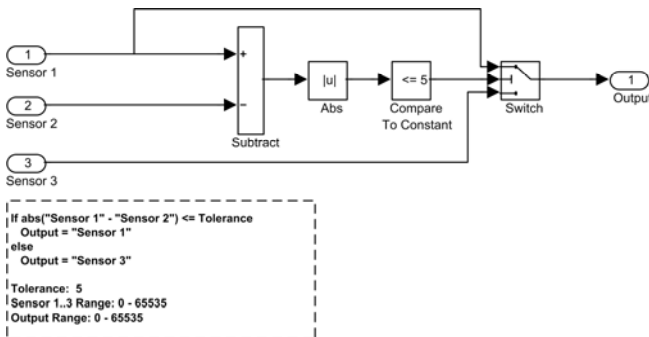


Fig. 1. Simulink model of a sensor voter

Figure 1 shows a TMR (triple modular redundancy) system which detects and masks a single faulty sensor value. Three input values are received from corresponding sensors and if one sensor is faulty, the output is assigned the value of one of the two fault-free sensor values.

When MODIFI injects a fault during runtime, i.e. a fault-injection block is automatically inserted in the model, it injects the fault effect (denoted a Failure Mode Function, FMF) before the execution occurs, and also adds an additional input port that controls the triggering of the fault (see Figure 2).

The FMF library is implemented in XML using the proposed schema described in this paper. An example of fault injection with MODIFI is depicted in Figure 2, showing how an FMF block is inserted into a single location (Sensor 1 input block out port) in the previously described Simulink model to simulate the occurrence of a hardware fault. The Simulink artifact that is inserted into the model is a block which encapsulates the Matlab m-code that is automatically generated from the FMF library XML-file (further described in following sections).

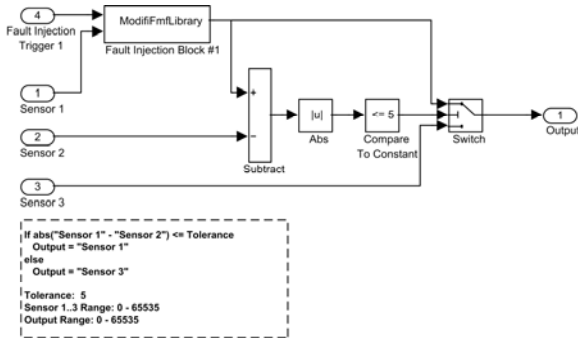


Fig. 2. Simulink model of a sensor voter with an FMF block inserted

During the configuration of the MODIFI tool, the selection of FMFs that shall be used for each available fault injection location in the model is made manually (see Figure 3), similar to preparing an FMEA for a safety-critical embedded system. Only FMFs that are actually applicable for the artifact are shown in the GUI. In the example depicted, the data type of the selected signal is uint16, i.e. 16 bit unsigned integer. Thus only FMFs that are applicable to unsigned integers are displayed (Flip bits FMF in the depicted example).

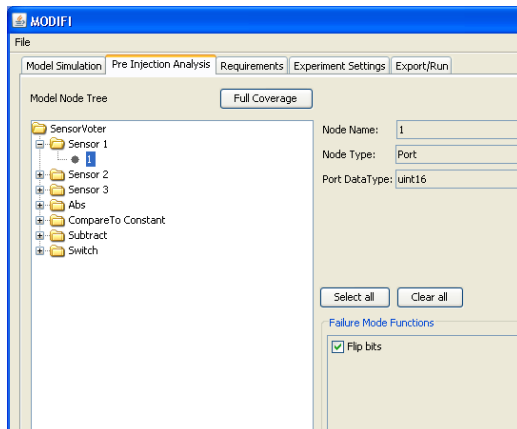


Fig. 3. MODIFI GUI – Assignment of fault models to fault injection locations

During the fault injection campaign (a set of experiments), the user can follow the progress through the *progress cube* that shows the status of each experiment. Figure 4 shows the progress of a campaign for the sensor voter with 120 experiments, each represented as a box in the cube. A blue box denotes an experiment to be performed, a green box denotes an experiment that did not violate a safety requirements and a red box denotes an experiment where safety requirements were violated. The bit-flip fault model is used (Fmf: 30) and the six locations corresponds to the three inputs and three internal signals of the model. In this demonstration example, the model is only run for four time steps.

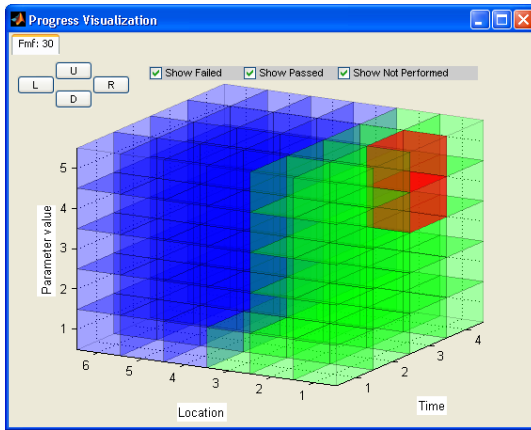


Fig. 4. MODIFI progress cube visualization

3 Fault Models

A fault is defined as a time-location pair, meaning that two faults injected at the same location, but at two different instants in time, could have a completely different effect on the system. Thus, the time aspects are important and faults are thus injected during execution of the target system. A fault model (also denoted as a failure mode in e.g. safety standards) can be defined by means of the number of faults, the time aspect and the fault type. An example of a fault model is a *single transient bit-flip fault*.

3.1 Failure Modes

Traditionally, fault injection is used to simulate hardware faults. The term *hardware fault* refers to a physical fault. We emphasize that hardware faults can be simulated by using fault injection mechanisms implemented in the model (MIFI), implemented by extra software (SWIFI), or implemented by using extra hardware (HIFI). Furthermore, hardware faults can be simulated at different levels of abstraction. A distortion of a signal may be caused by a bit-flip fault at the gate-level (i.e., in silicon), or by electromagnetic interference disturbing a transmission channel.

3.2 Failure Mode Functions

An FMF can be seen as an implementation of a fault effect, which is utilized to manipulate signals between blocks in a model, or that manipulates operators, to simulate

the effect of faults/errors that will lead to a failure (e.g. a value failure or a component failure). A general remark valid for all FMFs presented in this section is that at each time step when a failure is not activated the *actual value* is equal to the *nominal value* (non-faulty value); else the actual value assumes the value determined by the FMF. The temporal aspect (transient, intermittent, permanent) of the FMF is defined by a boolean input to the FMF where a true value means that the FMF is activated.

Table 1 shows two fault models that are defined within the MOGENTES project; *Flip_bits* and *Set_bits*. *Flip_bits* simulates the effect of a bit flip fault, while *Set_bits* simulates the effect of e.g. stuck-at faults.

Table 1. Bit-level failure mode function description

Name	Description
Flip_bits	This FMF will flip (alter the logical value for) one or more bits in the representation of the actual value.
Set_bits	This FMF will set one or more bits to a specific bit pattern in the representation of the actual value.

4 Modeling of Fault Models

The purpose of the proposed XML schema is to provide an unambiguous, implementation independent description format from which implementations can be automatically created (e.g. by code generation). The overall design goal is that the modeling of the more than 30 fault models that have been defined within the MOGENTES project shall be possible with this format.

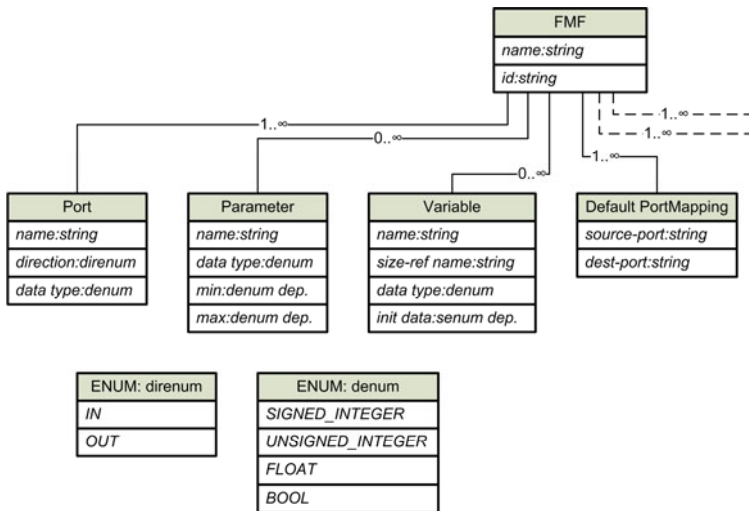


Fig. 5. FMF model part 1

Figure 5 shows the left part of the fault model description format while Figure 6 shows the right part. A failure mode function is distinguished through its *id* parameter, which shall be a unique identifier. The *name* parameter on the other hand is an arbitrary short-name, which not needs to be unique. As an example, the FMFs with ids “30u”, “30s” and “30f” are all named “Flip bit”, but for different data types (unsigned integer, signed integer and floating point value, respectively). In MODIFI, only fault models that are applicable for the data-type of a specific location are visible in the configuration GUI. Therefore, only one of the three fault models is visible. The data-types that are supported in this format specification are: unsigned integer, signed integer, floating point value and boolean.

To support a wide set of fault models, the interface to a failure mode function is defined using ports and parameters. Ports are specified by an FMF unique *name*, a *direction* (IN or OUT) and a *data type* (UNSIGNED INTEGER, SIGNED INTEGER, FLOAT or BOOLEAN). Parameters are defined similarly (without *direction*) with the addition of *min* and *max* value of the parameter value. These are used e.g. when the fault injection tool (e.g. MODIFI) is able to choose the value of a parameter, e.g. which bit to flip for the previous “flip bits” example.

The FMF description also supports variable allocation of dynamic length, e.g. to store port values during several time steps for produced-too-late faults.

To know which input ports and output ports that shall be connected by default, *default port-mappings* can be added to the description. These are typically used to map a single output to a single input.

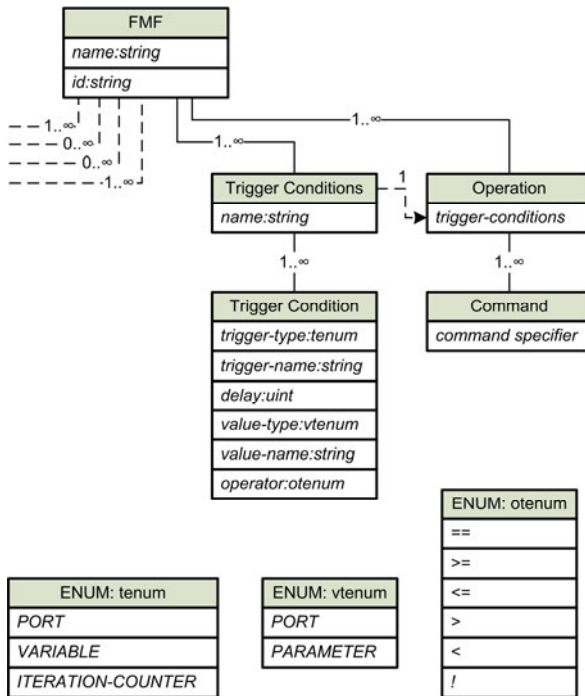


Fig. 6. FMF model part 2

Figure 6 depicts the actual behavior specification of a failure mode function. An FMF contains at least one *operation*, which in turn contains a command sequence of at least one *command* that is executed if the referenced *trigger-conditions* are fulfilled. The *command specifier* is one of the commands in Table 2, which shall be supported by a generator that conforms to the proposed FMF library format.

Table 2. Mandatory commands for a FMF library

ASSIGN_OUT	<i>Copy a specified input to a specified output.</i>
ASSIGN_OUT_BOOL_NOT	<i>Copy a specified Boolean input to a specified boolean output, but invert the value.</i>
ASSIGN_OUT_BITWISE_OR	<i>Copy a specified input to a specified output, but do a bitwise OR with the specified value.</i>
ASSIGN_OUT_BITWISE_XOR	<i>Copy a specified input to a specified output, but do a bitwise XOR with the specified value.</i>
ASSIGN_OUT_BITWISE_AND	<i>Copy a specified input to a specified output, but do a bitwise AND with the specified value.</i>
ASSIGN_OUT_RANDOM	<i>Copy a random value (between a min and max value) to a specified output.</i>
ASSIGN_OUT_MULT	<i>Copy the multiplication between two inputs to a specified output.</i>
ASSIGN_OUT_ADD	<i>Copy the addition between two inputs to a specified output.</i>
ASSIGN_OUT_SUB	<i>Copy the subtraction between two inputs to a specified output.</i>
FILL_VARIABLE	<i>Fill a variable with a specified input.</i>
SHIFT_VARIABLE	<i>Shift a variable (shift in a specified input).</i>

A *trigger-conditions* container is identified by a unique *name*, and contains an arbitrary number of *trigger-condition* containers which all shall be evaluated to TRUE for the *trigger-conditions* container evaluation to be evaluated as TRUE.

A *trigger-condition* container is defined as follows. A *trigger-type* (PORT value, VARIABLE value or the ITERATION-COUNTER value) and *trigger-name* is used to identify the left-hand side, and a *value-type* (PORT value or PARAMETER value) and *value-name* is used to identify the right-hand side of the evaluation of the *operator* (==, >=, <=, >, <, ! (i.e. NOT)).

The following example shows how the effect of a bit-flip fault (Flip_Bits FMF) is modeled using the proposed format, with the addition of a trigger to enable/disable the fault effect. This is useful to support different persistence models.

Table 3. FMF description of bit-flip fault

PORTS <i>{name, direction, data type}</i>	{Input1, IN, SIGNED_INTEGER} {Trigger, IN, BOOL} {Output1, OUT, SIGNED_INTEGER}
PARAMETERS <i>{name, data type, min, max}</i>	{Bit, UNSIGNED_INTEGER, 0, 31} {True, BOOL, true, true}
VARIABLES <i>{name, data type, size source, size name, init value}</i>	
DEFAULT-PORT-MAPPINGS <i>{source port, destination port}</i>	{Input1, Output1}
TRIGGER-CONDITIONS <i>{name, {trigger type, trigger name, delay, value type, value name, operator}}</i>	{TrigCond, {PORT, Trigger, 0, PARAMETER, True, ==} }
OPERATIONS <i>{trigger-name, {command specifier, PARAMETERS}}</i>	{TrigConds, {ASSIGN_OUT_BITWISE_XOR, Output1, Input1, Bit} }

Using a C source code generator on the description in Table 3, the source code in Figure 7 is produced.

```

/***** Failure Mode Function 30s (Flip bits) *****/
unsigned int Fmf30s_Bit;
bool Fmf30s_True;
unsigned long Fmf30s_UpdateCounter;

static void Fmf30s_Init(unsigned int Bit, bool True)
{
    Fmf30s_Bit = Bit;
    Fmf30s_True = True;
    Fmf30s_UpdateCounter = 0;
}

static void Fmf30s_Update(int Input1, bool Trigger, int *output1)
{
    *output1 = Input1;

    // Operation 1
    if((Trigger == Fmf30s_True))
    {
        // Command: ASSIGN_OUT_BITWISE_XOR
        *output1 = Input1^Fmf30s_Bit;
    }
}

```

Fig. 7. Generated C source code from FMF description in Table 3

For usage with the MODIFI tool, a Matlab [9] m-code generator has been created that produces Matlab-executable code, similarly to the previous example. This is depicted in Figure 8.

```
%----- Failure Mode Function 30s (Flip bits) -----
function Fmf30s_Init(Bit, True)
    FmfVars = evalin('base', 'FmfVars');
    FmfVars.Fmf30s_Bit = Bit;
    FmfVars.Fmf30s_True = True;
    FmfVars.Fmf30s_UpdateCounter = 0;
    assignin('base', 'FmfVars', FmfVars);
return;

function output = Fmf30s_Update(Input1, Trigger)

    FmfVars = evalin('base', 'FmfVars');
    output.Output1 = Input1;

    % Operation 1
    if((Trigger == FmfVars.30s_True))
        % Command: ASSIGN_OUT_BITWISE_XOR
        output.Output1 = bitflip(Input1, FmfVars.Fmf30s_Bit);
    end;

    FmfVars.Fmf30s_UpdateCounter = FmfVars.Fmf30s_UpdateCounter + 1;
    assignin('base', 'FmfVars', FmfVars);
return;
```

Fig. 8. Generated m source code from FMF description in Table 1

In MODIFI, the generated code is encapsulated into a Simulink block that is inserted automatically by the fault injection engine into the model, as shown in Figure 2. Parameters values, that are realized as input parameters to the FMF initialization function (e.g. `Fmf30s_Init` in Figure 8) are randomly chosen by the fault injection engine to a value between the minimum and maximum values defined in the FMF library. In the previous example, a value between 0 and 31 will be chosen for the *Bit* parameter, while the value `true` will be chosen for the *True* parameter. The latter is a way of defining constants, i.e. by setting the minimum and the maximum value of a parameter to the same value.

5 Conclusions

To support interoperability between fault injection tools, an XML schema based format has been proposed for describing effects of hardware faults, denoted failure mode functions (FMF's). An FMF is described by using the provided properties (e.g. trigger conditions, operations, ports and parameters) based on the anticipated effect of a hardware fault occurring in a system, similar to the preparation of an FMEA for a safety-related electronic system.

Code generators have been implemented for both C code and Matlab code. These generators take an FMF library in XML-format and produce code for each FMF (examples depicted in Figure 7 and Figure 8) together with a generic part that act as an interface between the fault injection engine and the library.

These generators and the proposed XML schema have been used successfully by fault injection tools that were developed within the MOGENTES project.

Acknowledgments. This work has been funded by the research project MOGENTES within the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement no 216679.

References

1. Svenningsson, R., Vinter, J., Eriksson, H., Törngren, M.: Model-Implemented Fault Injection for Hardware Fault Simulation. In: Proceedings of the Workshop on Model-Driven Engineering, Verification, and Validation (MoDeVVa 2010), Oslo, Norway, October 3, pp. 31–36 (2010)
2. The MOGENTES Project, <http://www.mogentes.eu>
3. IEC International Electrotechnical Commission, IEC 60218:2006 Analysis Techniques for System Reliability – Procedure for Failure Mode and Effects Analysis (FMEA)
4. Svenningsson, R., Vinter, J., Eriksson, H., Törngren, M.: MODIFI: A MODEL-Implemented Fault Injection Tool. In: Schoitsch, E. (ed.) SAFECOMP 2010. LNCS, vol. 6351, pp. 210–222. Springer, Heidelberg (2010)
5. IEC International Electrotechnical Commission, IEC 61025:2006 Fault Tree Analysis (FTA)
6. IEC TR 62380 – Reliability data handbook, <http://www.iec.ch>
7. W3C XML Schema, <http://www.w3.org>
8. da Silva, A., et al.: XML schema based fault set definition to improve fault injection tools interoperability. *International Journal of Critical Computer-Based Systems* 1(1/2/3), 220–237 (2010)
9. Mathworks, <http://www.mathworks.com>

Tightening Test Coverage Metrics: A Case Study in Equivalence Checking Using k -Induction*

Alastair F. Donaldson¹, Nannan He¹, Daniel Kroening¹, and Philipp Rümmer²

¹ Computer Science Department, Oxford University, UK

² Uppsala University, Department of Information Technology, Uppsala, Sweden

Abstract. We present a case study applying the k -induction method to equivalence checking of Simulink designs. In particular, we are interested in the problem of equivalence detection in mutation-based testing: given a design S , determining whether a “mutant” design S' derived from S by syntactic fault injection is behaviourally equivalent to S . In this situation, efficient equivalence checking techniques are needed to avoid redundant and expensive search for test cases that observe differences between S and S' . We have integrated k -induction into our test case generation framework for Simulink. We show, using a selection of benchmarks, that k -induction can be effective in detecting equivalent mutants, sometimes as a stand-alone technique, and sometimes with some manual assistance. We further discuss how the level of automation of the method can be increased by using static analysis to derive strengthening invariants from the structure of the Simulink models.

1 Introduction

Mutation-based testing [20] is an effective technique for generating high-quality test suites for software systems. The technique is based on the hypothesis that a test capable of detecting a small, synthesized error in a program may very likely be able to detect real defects introduced accidentally by programmers. Applying mutation-based testing typically works as follows. A set of *mutations* is identified. These are small syntactic changes to the system under test. An example mutation might be the replacement of operator $+$ by $*$ at a particular program point, or the modification of a signal in a dataflow program by injecting a new computation block. Then, a search is carried out to find a set of test cases that *kill* many of the mutants: a test case kills a mutant if it exposes the fact that the behaviours of the original and mutated system diverge. The more mutants a test suite kills the higher the coverage of the suite, while the smaller the test suite the more efficiently it can be executed during software development. Given a sufficiently rich set of mutation operators, mutation coverage subsumes many

* This research is supported by the EU FP7 STREP MOGENTES (project ID ICT-216679), the ARTEMIS CESAR project, the EU FP7 STREP PINCETTE (project ID ICT-257647), and EPSRC grant EP/G051100.

other popular notions of coverage, such as location coverage and MC/DC for software and stuck-at faults for hardware [24].

Ideally, we would like to be able to efficiently derive a small set of test cases that kill all of a given set of mutants. A problem with this ideal is the possibility of *equivalent mutants*. Because mutants are obtained in a lightweight, syntactic fashion, there is no guarantee that a given mutant will in fact exhibit different behaviour from the original system. If no such difference can ever be observed, we say that that mutant is *equivalent* to the original system. Clearly, no test case can ever kill an equivalent mutant. We do not wish to waste time attempting to derive such test cases, and it would be unfair to regard a test suite to be of low quality because it does not kill mutants that are actually equivalent. Thus there is a need for techniques to automatically detect equivalence of mutants.

In previous work [5] we proposed a mutation-based test case generation approach for Simulink models. The basic idea is to inject multiple mutants into a Simulink model to obtain a mutated design. The original and mutated designs are automatically translated into an ANSI-C program that runs the designs in-step, asserting that they are observationally equivalent. Then, bounded model checking (BMC) [3] using a tool such as CBMC [6] is performed to check whether the designs really are observationally equivalent up to a given execution depth. Bounded equivalence is checked using a SAT solver like MiniSat, such that non-equivalence is detected when the solver finds a satisfying assignment to the associated SAT problem. Such satisfying assignments are used to derive test cases to kill the injected mutants.

While effective, this BMC-based approach is expensive, thus we would like to avoid applying it to injected mutants which are equivalent. In this work, we present a case study using the k -induction method [26] for equivalence checking in the Simulink domain. We show, using a selection of benchmarks, that k -induction can be effective in detecting equivalent mutants, sometimes as a stand-alone technique, and sometimes with some manual assistance. We further discuss how the level of automation of the method can be increased by using static analysis to derive strengthening invariants from the structure of the Simulink models.

2 Mutation-Based Test Case Generation for Simulink

2.1 Matlab Simulink

Matlab Simulink is a graphical dataflow language that is commonly used in industry for modeling or implementing control applications. Simulink models consist of a set of *blocks* that are connected by *signals* specifying the flow of data. Blocks are taken from pre-defined block libraries (covering generic functions such as addition or logical operators, but also domains like fuzzy logic or network communication) and receive a specific number of input signals from which output signals are computed. Stateful systems are modeled with the help of feedback loops. Models can be structured hierarchically with the help of *subsystems*, and

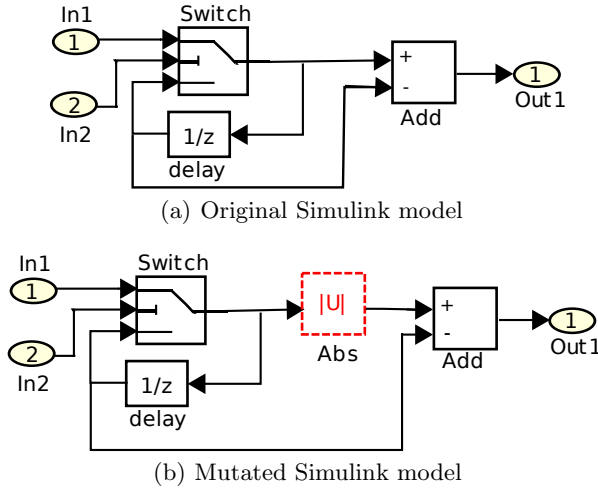


Fig. 1. Example of a mutated Simulink model

can be simulated, analyzed, or compiled to code using the Matlab tool-suite and third-party products.

For the purposes of this paper, we only consider discrete-time Simulink models, which means that signals represent (potentially infinite) streams of values governed by a global clock. The semantics of blocks is synchronous in the sense that every block is evaluated and performs exactly one computation step per time unit. As a whole, a Simulink program receives a number of (potentially infinite) streams of input values (specified using *inports* in the Simulink model) and generates a number of output streams (described using *outports*).

An example of a Simulink model is given in Fig. 1(a). This model has two inports $In1$, $In2$, and one outport $Out1$. The inputs are connected to a *Switch* (a multiplexer) that forwards either $In1$ or the output of *delay*, depending on the value of $In2$. The unit-delay block *delay* is responsible for storing the output of *Switch* for one time unit, thus preventing a cyclic definition caused otherwise by the feedback loop from *Switch* to itself. All instances of the unit-delay block must be initialized. The default initialization value is 0. The block *Add* (actually performing subtraction) computes the difference between the results of *Switch* and *delay* and feeds it to the outport $Out1$.

2.2 Mutation-Based Test Case Generation

In this paper, we consider test case generation (TCG) strategies for Simulink models built on top of the mutation-based TCG approach defined in [5], which uses bounded model checking techniques to systematically construct test cases. *Mutation-based TCG* proceeds by injecting syntactic mutations (in this context sometimes also called *faults*) into a given Simulink model S , generating from S a set M of *mutants*. We use S' to refer to a mutant in M .

Fig. 1 is an example mutant derived from the Simulink model of Fig. 1(a). In the mutant, the output of *Switch* is replaced by its absolute value before being input to *Add*. Assume that an input has the form $(In1, In2)$, where *In2* causes the top input of *Switch* to be selected if it is 1, otherwise the bottom input is to be selected; and initial outputs for all unit delay blocks are 1. An example input sequence $\langle\langle 1, 1 \rangle, \langle -1, 1 \rangle\rangle$ leads to the output sequence of Fig. 1(a) as $\langle 0, -2 \rangle$, while applied to Fig. 1(b), the output sequence is $\langle 0, 0 \rangle$ (recall that, as discussed in §2.1, the *Add* blocks of Fig. 1 actually perform subtraction). The difference in final values for these output sequences, highlighted in bold, indicates that the models behave differently.

The goal of TCG is to find a set of test cases (finite sequences of inputs for the model S) that *kill* each of the mutants in M , which means that the test case makes a mutant $S' \in M$ produce outputs that differ from those of the original model S . For example, as argued above, the test case $\{\langle\langle 1, 1 \rangle, \langle -1, 1 \rangle\rangle\}$ kills the mutant of Fig. 1. The main hypothesis underlying mutation testing is that such test cases, which are able to detect simple bugs like the injected syntactic mutations, are also useful for finding real, potentially more complicated defects (this is called the *coupling effect* [9]).

In the style of bounded model checking [2], both the original model S and each of its mutants $S' \in M$ can abstractly be modeled using transition relations T and T' and formulae I, I' defining the initial states. Like in equivalence checking [21], observational equivalence of S and S' during the first d computation steps can then be expressed using the following formula:

$$\begin{aligned}
 & \underbrace{I(s_0) \wedge \bigwedge_{i=0}^{d-1} T(s_i, s_{i+1})}_{\text{first model}} \wedge \underbrace{I'(s'_0) \wedge \bigwedge_{i=0}^{d-1} T'(s'_i, s'_{i+1})}_{\text{second model}} \wedge \underbrace{\bigwedge_{i=0}^d s_i.i = s'_i.i}_{\text{equality of all inputs}} \\
 \Rightarrow & \underbrace{\bigwedge_{i=0}^d s_i.o = s'_i.o}_{\text{equality of all outputs}}
 \end{aligned} \tag{1}$$

Any countermodel to this formula represents two executions of S and S' that yield a different output sequence; the projection of the assignment to the inputs corresponds to a test case. As most Simulink models operate on scalar datatypes such as integers or floating-point arithmetic, and therefore have a finite state space, countermodels can be constructed using SAT/SMT-based techniques.

2.3 From Simulink to C: Our Test Case Generation Tool Chain

Due to the complexity of the Simulink language and the size of commonly used block type libraries, the development of analysis tools directly operating on Simulink models is a huge effort. We therefore follow a compilation approach and convert Simulink models to C programs prior to test case generation. Further processing can then be performed by ANSI-C analysis tools, in our case based on

```

// Declaration of inputs
signal_type in0, in1, ...;
// Declaration of internal signals
signal_type sig0, sig1, ..., sig0_m, sig1_m, ...;
// Declaration of outputs
signal_type out0, out1, ... out0_m, out1_m, ...;

int main () {
// The main simulation loop
  for (sim_time=START; sim_time<END; sim_time+=sim_step) {

// Reading inputs
    in0 = readInput0(); in1 = readInput1(); ...

// Execution of the original model S
    sig0 = ...; sig1 = ...; ...
    out0 = ...; out1 = ...; ...

// Execution of the mutant S'
    sig0_m = ...; sig1_m = ...; ...
    out0_m = ...; out1_m = ...; ...

// (*)
  }
}

```

Fig. 2. Skeleton of C code generated from Simulink models

the CBMC [6] bounded model checker. For the compilation from Simulink to C, we use two different tools: our own Simulink front-end [5], which is tightly integrated with CBMC and optimised for static analysis (applied to the generated C programs); and Gene-Auto [27], an industrial-grade open-source code generator for Simulink.

In our experiments, mutations are always applied at the level of Simulink models (rather than, as would also be possible, at the level of the C code generated from a Simulink model):

1. a given Simulink model S is first duplicated (cloned) by creating a copy S' of S connected to the same input ports as S .
2. the clone S' is mutated by inserting a further computation block into one of the signals of S' . The mutation operators considered in this paper are described in Sect. 5; the work presented here directly generalises to further mutation operators.

Examples of the resulting models are given in Fig. 5.

Compiling Simulink models to C results in programs of the structure shown in Fig. 2. The type `signal_type` will practically be either `int` or `float` (in our experiments, the former). Note that the program can contain multiple, nested

```
assert(out0 == out0_m && out1 == out1_m && ...); // (*)
```

Fig. 3. Assertion relating the outputs of the original and mutated Simulink models. Counterexamples that violate this assertion provide test vectors that kill the mutant

loops in addition to the main simulation loop, since the code generated for the models S and S' might itself contain loops. Loops other than the main simulation loop are, however, usually bounded.

In order to generate test cases, an assertion relating the different outputs is added at (*) in Fig. 2; the added assertion is shown in Fig. 3. Counterexamples demonstrating that the assertion (*) can be violated represent test cases killing the considered mutant. Such counterexamples can effectively be constructed using bounded model checkers such as CBMC [6]. The tool COVER [5] automates this process and produces test cases in an XML-based format.

2.4 The Phenomenon of Equivalent Mutants

Not all mutations give rise to observably different behaviour of a model. In fact, one of the main obstacles in traditional mutation testing is the difficulty of identifying mutations that do not have an observable effect on system outputs. Suppose formula (I) from §2.2 is valid for some d , which means that the applied mutation does not result in an error that propagates to an observable output within d steps. There are two possible reasons for this:

1. The bound d is not sufficiently large to reveal the error.
2. The model contains redundancy and the mutation does not result in an observable change of its behaviour. In other words, (I) is valid *for any* d . The mutant is in this case called an *equivalent mutant*.

The first case could be addressed by simply increasing the bound d . However, bounded model checking alone is not sufficient to distinguish between the two cases, since there is no upper bound (or only prohibitively large bounds, taking the usually finite state space of a Simulink program into account) on the values of d that have to be considered. In order to detect case 2, it is therefore necessary to apply techniques beyond bounded model checking; the approach evaluated in this paper is based on strong versions of *induction* and inductive invariants.

3 Detection of Equivalent Mutants Using k -Induction

The k -induction method was proposed as a technique for SAT-based verification of finite-state transition systems [26], and has been used successfully to verify complex hardware designs, in particular pipelined architectures. Recently, k -induction has also been applied in the verification of imperative software [12,13]. In this paper, we consider applying the software formulation of k -induction proposed in [12,13] to detect mutant equivalence in the C programs generated via the technique described in Sect. 2.3.

3.1 k -Induction for Transition Systems

Let $\mathbf{I}(x)$ and $\mathbf{T}(x, y)$ be formulae encoding the initial states and transition relation for a system over sets of state variables x and y , $\mathbf{P}(x)$ a formula representing states satisfying a safety property, and k a non-negative integer. To prove \mathbf{P} by k -induction one must first show that \mathbf{P} holds in all states reachable from an initial state within k steps, i.e., that the following formula (the base case) is unsatisfiable:

$$\mathbf{I}(s_1) \wedge \mathbf{T}(s_1, s_2) \wedge \cdots \wedge \mathbf{T}(s_{k-1}, s_k) \wedge (\overline{\mathbf{P}(s_1)} \vee \cdots \vee \overline{\mathbf{P}(s_k)}) \quad (2)$$

Secondly, one must show that whenever \mathbf{P} holds in k consecutive states s_1, \dots, s_k , \mathbf{P} also holds in the next state s_{k+1} of the system. This is established by checking that the following formula (the step case) is unsatisfiable:

$$\mathbf{P}(s_1) \wedge \mathbf{T}(s_1, s_2) \wedge \cdots \wedge \mathbf{P}(s_k) \wedge \mathbf{T}(s_k, s_{k+1}) \wedge \overline{\mathbf{P}(s_{k+1})} \quad (3)$$

Having proved both the base and step case, we can conclude that \mathbf{P} holds in every reachable state of the transition system. The method can be made complete for finite-state systems by restricting the step case to consider only loop-free paths [26].

3.2 k -Induction in Mutation-Based Testing

In the context of mutation-based testing, we can use k -induction in order to show that (II) holds for any value of d . In this case, (II) forms the base case for k -induction: for some $d \geq 0$ we show that the original and mutated systems are equivalent up to depth d .

To obtain a complete result, we must also prove a step case as follows:

$$\underbrace{\bigwedge_{i=0}^d T(s_i, s_{i+1})}_{\text{first model}} \wedge \underbrace{\bigwedge_{i=0}^d T'(s'_i, s'_{i+1})}_{\text{second model}} \wedge \underbrace{\bigwedge_{i=0}^d s_{i.o} = s'_{i.o}}_{\text{equality of first } d \text{ outputs}} \quad (4)$$

$$\Rightarrow \underbrace{s_{d+1.o} = s'_{d+1.o}}_{\text{equality of output } d+1}$$

The step case ascertains that, given that the original and mutated systems have exhibited equal outputs for d steps, they are guaranteed to show equivalent outputs for a further step.

3.3 k -Induction for Software Programs

In prior work [12][13] we investigated a direct lifting of k -induction from transition systems to the level of program loops, in order to prove partial correctness of software programs with respect to assertions appearing in the program text.

Because we translate Simulink designs into C, it is this formulation of k -induction, rather than the transition system-level formulation outlined in §3.1 and §3.2, that we use to implement k -induction for detection of equivalent mutants. Our software k -induction method can directly be applied to programs such as the one shown in Fig. 2.

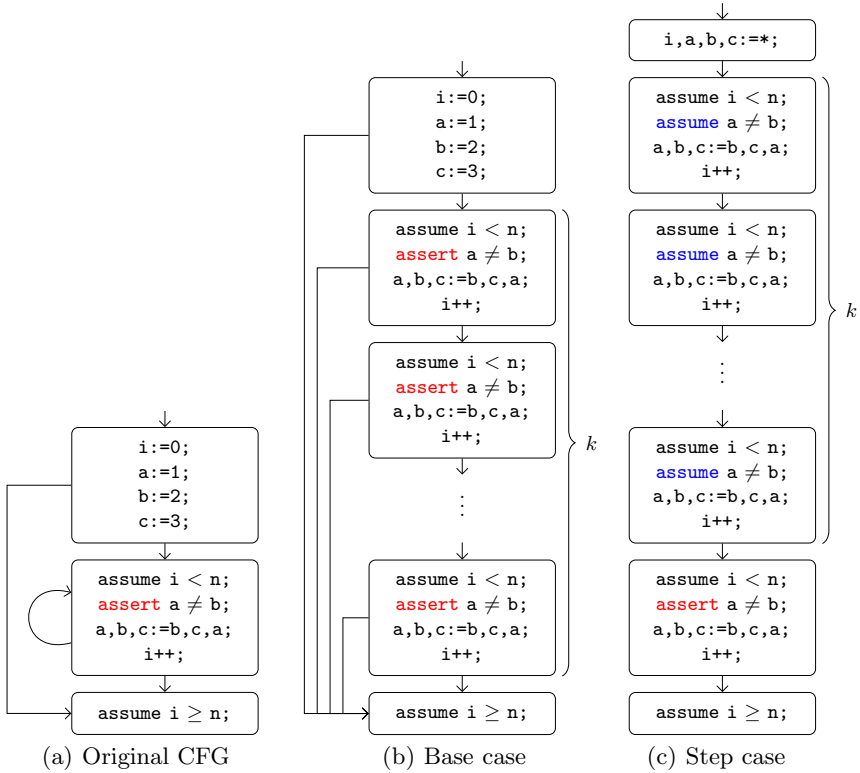


Fig. 4. A simple program, and the corresponding base and step cases for k -induction

The formal definition of our k -induction rule for programs is quite complex, but the intuition is simple. We shall explain the idea using an example, referring the reader to [12,13] for formal details.

We apply k -induction to a single loop in a program. Consider the example program of Fig. 4(a), depicted as a control flow graph (CFG), where flow of control is modelled using *assume* statements. (In particular, note that the loop condition $i < n$ is modelled by assuming that this expression holds on entry to the loop, and assuming that its negation holds on loop exit.)

We wish to prove that the assertion in the body of the loop can never be violated. This could be achieved using standard techniques by showing that $a \neq b \wedge a \neq c \wedge b \neq c$ is an inductive invariant for the loop, and that it implies the assertion $a \neq b$ of interest. However, with k -induction, we can prove this example correct *without* providing an external loop invariant. Instead, the assertion appearing in the loop body takes the role of an invariant.

From the CFG of Fig. 4(a), we derive two programs. The *step case program* (Fig. 4(c)) is analogous to (3). It checks whether, after executing the loop body

successfully k times from an arbitrary state, a further loop iteration can be successfully executed. In this further loop iteration, back edges to the loop header are removed, while edges that exit the loop are preserved. Thus the step case verifies that on loop exit, the rest of the program can be safely executed.

Because the program of Fig. 4(a) is indeed correct, the base case of Fig. 4(b) is correct for any $k \geq 0$. However, the step case of Fig. 4(c) is correct only for $k \geq 3$. To see that the step case does not hold for $k = 2$, consider the case where $n > 2$ and statement `i,a,b,c:=*` yields $i = 0$, $a = 1$, $b = 2$, $c = 1$. From this state, two loop iterations can be successfully executed, leading to a state where $a = 1$, $b = 1$ and $c = 2$, at which point the assertion $a \neq b$ does not hold.

It is this program-level approach to k -induction which we employ in order to detect equivalence of mutants in Simulink designs, applying induction to the C programs generated from our compilation flow.

4 Automatic Invariant Strengthening

We shall see in §5 that naïve application of k -induction is not strong enough to show equivalence of mutants in some typical cases. The intuitive reason why k -induction might fail is that the asserted property—that the outputs generated by the original model S and the mutant S' are equal—is not k -inductive for any k , since the resulting induction hypothesis gives too little information about the internal state of the Simulink programs. In general, it can be necessary to strengthen the invariant by adding conditions about the range of signals, or by equations asserting that signals of the original model and of the mutants carry the same value.

We examine two techniques to strengthen invariants automatically: abstract interpretation, using numeric abstract domains, and van Eijk's method to infer equalities between signals. We evaluate both techniques experimentally in §5.

4.1 Abstract Interpretation

In prior work [10] we have investigated ways to strengthen k -induction through static analyses, including abstract interpretation [7]. Given a control-flow graph to be analysed, suppose we use abstract interpretation (with some suitable domain) to determine that an invariant ϕ holds on entry to node n . Then, because abstract interpretation is a sound method, we can prepend the statement `assume(ϕ)` to n . In practice, we choose to prepend the statement `assert(ϕ)` rather than `assume(ϕ)`. This forces k -induction to re-check the inferred invariants, guarding against the possibility of vacuous results arising from bugs in the abstract interpreter.

By exploiting information about invariants in this way, we increase the possibility for k -induction to succeed in proving the property of interest: while the property may not be k -inductive in general, it may be k -inductive when restricted to the invariant obtained using abstract interpretation.

In §5 we discuss mutants that cannot be proven equivalent using k -induction alone, but for which equivalence can be proven if abstract interpretation, over the domain of intervals, is first used to compute a strengthening invariant.

4.2 Adaptation of van Eijk’s Method

The application of non-relational abstract domains (as in §4.1), for instance the interval domain, can significantly increase the proof strength of k -induction when detecting equivalent mutants. In this section, we propose a particular *relational* domain that further supports k -induction through eager computation of groups of signals that carry the same values in all executions of a Simulink model. The technique is inspired by van Eijk’s method [15], a method for sequential equivalence checking of hardware designs. In the original version, the method works by computing classes of signals that have the same (or opposite) values in all reachable states of a circuit; the computation is done using BDDs (to describe equivalence classes of signals and the transition relation of the circuit) and fixed-point iteration. Van Eijk’s method was combined with SAT-based verification, Stålmårck’s method, and k -induction in [4].

The need for relational information is illustrated in Fig. 6(a), in which the equivalence of the considered mutant can only be shown when adding the additional assertion that the output of the unit-delay block UD_m is not affected by the applied mutation. This is done by asserting that the outputs of the blocks UD and UD_m are equal. In this example, the use of a non-relational abstract domain alone is not sufficient for the equivalence proof. This situation is typical for Simulink programs with internal state that cannot completely be observed at the program outputs.

In general, we assume that a relation

$$R \subseteq \underbrace{\{\text{sig0}, \text{sig1}, \dots\}}_{\text{Sig}} \times \underbrace{\{\text{sig0}_m, \text{sig1}_m, \dots\}}_{\text{Sig}^m}$$

between original and mutated internal signals has been identified (where variables are named as in Fig. 2). The assertion inserted at (*) in Fig. 2 is then strengthened to:

```

    assert(out0 == out0_m && out1 == out1_m && ...); // (*)
{ assert(a == b); }_{(a,b) \in R}
```

Let us call this stronger set of assertions $A(R)$. If it is possible to verify $A(R)$ using k -induction, then also the original assertion, and thus the equivalence of the mutant has been proven.

In order to automatically compute relations R for which k -induction succeeds, we propose to first identify a set $R^c \subseteq \text{Sig} \times \text{Sig}^m$ of *candidate pairs* of signals. Natural candidates are pairs of corresponding signals in the original Simulink model and the mutant; such pairs are easy to compute and likely to carry the same values. Furthermore, the number of corresponding signal pairs is only linear

Algorithm 1. Iterative mutant equivalence checking

Input: C program as in Fig. 2, initial relation R^c , parameter $k \geq 0$
Output: One of {EQUIVALENT, NONEQUIVALENT, DONTKNOW}

```

 $R \leftarrow R^c$ ;
/* Eliminate signal pair candidates using random simulation */
repeat
  Execute Fig. 2 with assertions  $A(R)$  and random inputs;
  if assertion corresponding to pair  $(a, b) \in R$  failed then
    |  $R \leftarrow R \setminus \{(a, b)\}$ ;
  else if difference in outputs observed then
    | return NONEQUIVALENT;
  end
until timeout ;

/* Check  $k$ -induction base case */
repeat
  Check  $k$ -induction base case with assertions  $A(R)$ ;
  if assertion corresponding to pair  $(a, b) \in R$  failed then
    |  $R \leftarrow R \setminus \{(a, b)\}$ ;
  else if difference in outputs observed then
    | return NONEQUIVALENT;
  end
until base case succeeded ;

/* Check  $k$ -induction step case */
repeat
  Check  $k$ -induction step case with assertions  $A(R)$ ;
  if assertion corresponding to pair  $(a, b) \in R$  failed then
    |  $R \leftarrow R \setminus \{(a, b)\}$ ;
  else if difference in outputs observed then
    | return DONTKNOW;
  end
until step case succeeded ;

return EQUIVALENT;

```

in the size of the Simulink models. In some cases, it might, however, be beneficial to start from a larger set of signal pair candidates.

As second step, R^c is refined to a set $R^i \subseteq R^c$ by removing signal pairs that can be shown to have different values in some executions. This is done using two methods:

- by trying to verify the k -induction base case for the set $A(R^c)$ of assertions. Most likely, such a verification attempt will initially fail and report that some of the assertions in $A(R^c)$ could not be verified; the corresponding pairs have to be removed from R^i .
- by random simulation of the program in Fig. 2, using the set $A(R^c)$ of assertions. In practice, random testing can be expected to efficiently and quickly remove large numbers of candidate pairs from R^i .

The set R^i of remaining candidate pairs has to be refined by removing further signal pairs until the k -induction step case succeeds. This is done by trying to verify the k -induction step case with the set $A(R^i)$ of assertions; if some assertion of the step case cannot be verified, the corresponding pair $(a, b) \in R^i$ is removed, leading to the new relation $R^i := R^i \setminus \{(a, b)\}$. Iterating this procedure will eventually produce the greatest relation $R = R^i$ for which k -induction succeeds, or will terminate with the result that the equivalence of the considered mutant could not be proven.

Alg. [1](#) defines this technique more formally. The algorithm proceeds in three phases: 1. random simulation is used to remove as many signal pair candidates from R as possible; 2. the k -induction base case is verified, potentially ruling out further signal pairs in R ; and 3. the k -induction step case is verified, again reducing the set R as needed. For a given timeout bound for random simulation, the algorithm is guaranteed to terminate, and outputs as result either that the considered mutant was proven to be equivalent, that the mutant is non-equivalent (in which case it is also possible to extract a test case killing the mutant from the algorithm), or that the equivalence check was inconclusive.

Lemma 1 (Soundness of Alg. [1](#)). *If Alg. [1](#) returns the result EQUIVALENT (NONEQUIVALENT), the examined mutant is equivalent (not equivalent).*

Lemma 2 (Completeness of Alg. [1](#)). *If R^c contains a sub-relation $R^s \subseteq R^c$ such that k -induction is able to verify the assertions $A(R^s)$ for the C program in Fig. [2](#), then Alg. [1](#) returns the result EQUIVALENT when started with the initial relation R^c and the parameter k .*

Proof. By showing the following two properties: 1. for all relations R computed during the execution of the algorithm, it is the case that $R^s \subseteq R$; and 2. as long as $R^s \subseteq R$ during the execution of the algorithm, the result DONTKNOW is not returned. \square

It can be observed that Alg. [1](#) can be adapted also to other classes of assertions than just equations over signals; also the combination with the numeric abstract domains in Sect. [4.1](#) is straightforward.

5 Experiments

In this section, we first discuss four basic equivalent mutants extracted from larger, real-world benchmarks; we then report experimental results, and analyse two full-size examples in detail. To translate Simulink to C we use the tool presented in [5](#), as well as Gene-Auto [27](#). For equivalence checking, we use K-INDUCTOR [11](#), a prototypical version of CBMC extended with k -induction. For our adaptation of van Eijk's method ([84.2](#)), we have written a script which repeatedly invokes K-INDUCTOR to check base and step cases.

All experiments are performed on a computer with a 3 GHz Intel Xeon CPU and 48 GB of memory, running Linux.

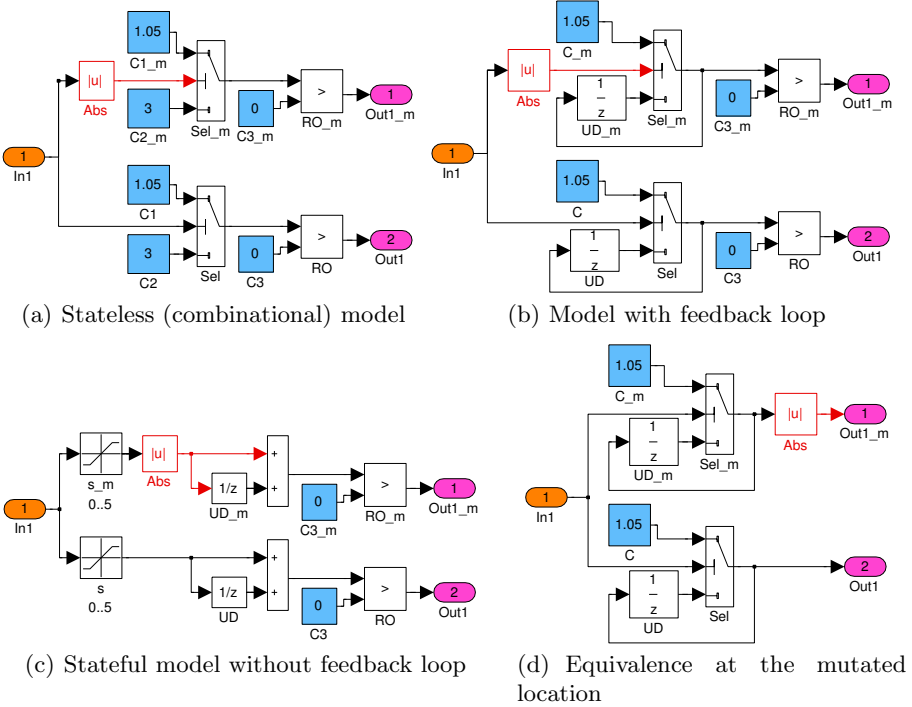


Fig. 5. Basic mutation scenarios occurring in our Simulink benchmarks

5.1 Simple Examples

In Fig. 5, each model consists of a mutant (the upper part) and the original model (the lower part). The mutant contains duplicates of all blocks in the original model; these blocks are distinguished by the suffix *_m*. The mutant and the original model share the same inputs. The outputs *Out1_m* and *Out1* are numbered 1 and 2 respectively, as they are distinct outputs in the overall Simulink model. The red blocks (named *Abs*) are the applied mutations, and thus appear in the upper part of each model. These are inserted absolute-value blocks. Blue blocks (whose name starts with *C*) are constants, labelled with their respective values. All blocks labelled with ‘ $1/z$ ’ are *Unit Delays* (memory blocks) and initialised with the value 1.

It can be observed that the outputs generated by the mutants and the original models coincide, no matter what the inputs to the models are, so that the mutants are indeed equivalent.

In case (a), the model does not include any state-related blocks or feedback loops, and is thus purely combinational. Using K-INDUCTOR, we could prove the equivalence of this mutant with $k = 1$.

Case (b) is more complex, since a *Unit Delay* block *UD* occurs after the mutated location. The input of this block is connected with the output of a *Switch*

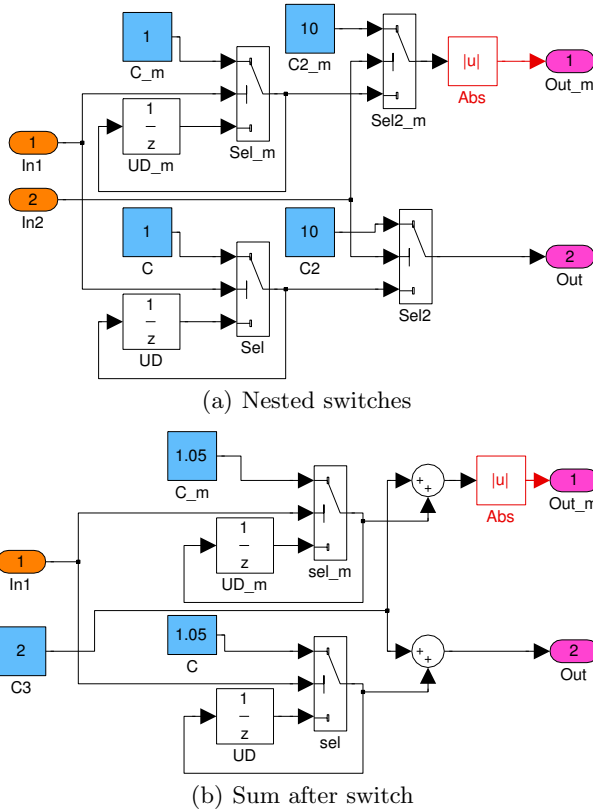


Fig. 6. Two cases of equivalences at mutated location

block *Sel*, carrying through either the upper or the lower input signal, depending on whether the value of the middle input is 0 or not. The output of *UD* is fed back to the switch *Sel*. This is a typical scenario observed in the benchmarks. Using *k*-induction alone, it was not possible to prove the equivalence of the mutant, since the step case could not be verified (for reasonable *k*). Verification is possible, however, when adding an assertion like `assert(Sel_m > 0 && Sel > 0)` into the generated C code, which makes the assertions in the simulation loop *k*-inductive for *k* = 2. Abstract interpretation over intervals, as discussed in §4.1, allows this assertion to be automatically derived.

Case (c) also includes a *Unit Delay* block, but is simpler because the output of the block *UD* is not fed back to its input. In this case, equivalence can directly be proven using *k*-induction with *k* = 2.

Case (d) is a special case where the mutation directly affects the output of the model. Also in this example, equivalence can directly be proven using *k*-induction with *k* = 2.

Two more interesting variants of this case are given in Fig. 6. We observe that *k*-Induction can directly prove Fig. 6(b), but not (a). In order to verify

equivalence in (a), it is necessary to add two further assertions to the simulation loop:

```
assert (UD_M == UD); assert (UD_M >= 0);
```

To obtain the first invariant, the adaptation of van Eijk’s method presented in §4.2 is used: the relation R^c is initialised with all pairs of signal variables occurring between the mutation point and outports. Algorithm 1 is then applied to remove invalid pairs. We did not employ random simulation, the first phase of Algorithm 1, to eliminate invalid pairs; we leave this to future work. Currently, elimination is performed solely by checking the k -induction base and step cases.

The second invariant can be derived automatically using abstract interpretation over intervals as discussed in §4.1

5.2 Larger Simulink Case Studies

In this case study, we make use of k -induction to check whether mutants injected into Simulink models are equivalent. We consider four Simulink benchmarks extracted from an industrial embedded software system used in the European MOGENTES project.¹ This software system contains control functions to implement steering anti catch-up in an automobile. We applied three different mutation operators to the Simulink models:

- ABS: Insert absolute value.
- UOI: Insert negation ($-$, \neg) operator.
- RR: Swap relational operators $<$, \leq , $>$, \geq , $=$.

The results of proving equivalence of mutants directly using k -induction, or after strengthening with added invariant assertions are summarised in Table 1. For every benchmark, *#Blocks* reports the total number of Simulink blocks (of the original model), while *#Muts* shows the number of generated mutants, each including exactly one mutation. *#S-blocks* gives the number of state-related blocks in the model (for example *Unit-Delay* blocks).

The *#Muts-RandT* column shows the number of mutants left after applying a simple random testing approach to kill mutants. *#K-Ind* and *#K-Ind-IS* give the number of mutants that are proved equivalent using our k -induction-based technique without and with invariant strengthening respectively (where strengthening uses abstract interpretation and our adaptation of van Eijk’s method). Note that the number for *#K-Ind-IS* is always larger than that for *#K-Ind*, because invariant strengthening can only increase the proof strength of our method.

The last column *#Ineq* reports the number of mutants which were not proved inequivalent using random simulation, but which were shown inequivalent by the base case of our k -induction approach. (Essentially, this means that these mutants can be shown to be inequivalent using shallow bounded model checking.)

In all cases, we find that $(\#K\text{-ind-IS} + \#Ineq) = (\#Muts\text{-RandT})$, thus our technique is able to fully categorise mutants as equivalent or inequivalent for this set of benchmarks.

¹ <https://www.mogentes.eu/>

Table 1. Summary of experimental results

<i>Benchmark</i>	<i>#Blocks</i>	<i>#Muts</i>	<i>#S-blocks</i>	<i>#Muts-RandT</i>	<i>#K-ind</i>	<i>#K-ind-IS</i>	<i>#Ineq</i>
CalcOffset	80	69	2	10	3	3	7
Decision	92	87	2	15	0	7	8
RecogLoc	66	40	1	11	0	5	6
Spoiler	44	36	0	5	5	5	0

Table 2. Experimental results of proved mutants

<i>ID</i>	<i>Benchmark</i>	<i>Mutation-Type</i>	<i>#T(s)</i>	<i>k</i>	<i>#Assert. v.E.</i>
M-1	Decision	ABS	1.21	2	N/A
M-2	Decision	ABS	3.23	2	3/3
M-3	Decision	ABS	4.15	2	5/5
M-4	Decision	ABS	4.34	4	6/1
M-5	Decision	ABS	1.23	3	N/A
M-6	Decision	ABS	3.24	3	6/4
M-7	Decision	RR	3.72	3	5/1
M-8	RecogLoc	ABS	3.81	4	5/2
M-9	RecogLoc	ABS	3.78	4	4/2
M-10	RecogLoc	ABS	4.74	4	6/4
M-11	RecogLoc	ABS	3.31	2	6/4
M-12	RecogLoc	RR	3.51	2	6/4

The *CalcOffset* benchmark includes two *Unit Delay* blocks. The outputs of these blocks are not fed back to their inputs, similar to Fig. 5(c). The experiments show that k -induction alone is capable of proving equivalent mutants, and the runtime of each proof is within 1 second. For the benchmarks *Decision* and *RecogLoc*, invariant strengthening indeed increases the number of mutants that k -induction can prove equivalent. The *Spoiler* benchmark model has no state-related blocks (similar to Fig. 5(a)); all potentially equivalent mutants are indeed proven equivalent by k -induction alone within 1 second.

Table 2 gives detailed information about mutants injected into the *Decision* and *RecogLoc* models. As shown in Table 1, none of these mutants could be shown equivalent by k -induction alone—all required invariant strengthening. The column *#T(s)* gives the runtime needed for verification, in seconds. It includes the total time for running our abstract interpreter, followed by the van Eijk-based method of Algorithm 1 if needed (which may call K-INDUCTOR multiple times). The column k gives the parameter k required to prove equivalence of a mutant. The last column shows the number of signal-equivalence invariant assertions which were added by the van Eijk-based method of Algorithm 1, if this strengthening is necessary. For example, for the mutant *M-1*, “N/A” means that k -induction strengthened with abstract interpretation is sufficient to prove the equivalence, without invoking our adaptation of van Eijk’s method. In contrast, for mutant *M-6*, “6/4” reports that six potential assertions are identified by the van Eijk technique, and four of them are proved to be invariants and

used in strengthening k -induction. Overall, the experiments show that most of these equivalent mutants (10 out of 12) require strengthening with both abstract interpretation and our adaptation of van Eijk’s method in order for k -induction to succeed. We did not find any cases among the *Decision* or *RecogLoc* mutants where the van Eijk technique allowed equivalence to be proven without abstract interpretation also being applied.

6 Related Work

Mutation-based test case generation. The concept of mutation testing was first introduced in 1971 in Richard Lipton’s class term paper “Fault diagnosis of computer programs.” Since then, it has become a standard method for evaluating the quality of test suites and been applied to software systems of considerable size, see [20] for a broad survey. In this paper, we only consider mutant models with single mutations, whereas other authors also consider combinations of faults [22]. In [25], Ruthermel et al. propose to use mutations to prioritise test cases to increase a test suite’s rate of fault detection.

This work is partly based on our framework [5] for test case generation for Simulink. An optimisation of the framework, applying formal concept analysis to cluster mutants, has been described in [19].

k-Induction. The concept of k -induction was first published in [26,4], targeting the verification of hardware designs represented by transition relations (although the basic idea had already been used in earlier implementations [23] and a version of one-induction used for BDD-based model checking [8]). A major emphasis of these two papers is on the restriction to loop-free or shortest paths, which is so far not considered in our k -induction rule due to the size of state vectors and the high degree of determinism in software programs. Several optimisations and extensions to the technique have been proposed, including property strengthening to reduce induction depth [28], improving performance via incremental SAT solving [14], and verification of temporal properties [1].

Besides hardware verification, k -induction has been used to analyse synchronous programs [18,16], SystemC designs [17] and imperative software [12,13]. A combination of the k -induction rule of [12,13], abstract interpretation, and domain-specific invariant strengthening techniques for race analysis is the topic of [10].

7 Conclusions and Future Work

We have presented a case study in the application of k -induction to the problem of detecting equivalent mutants in mutation-based test case generation for Matlab Simulink. Our experiments show that k -induction shows promise in this area, proving successful in equivalence detection for a range of examples, sometimes as a stand-alone technique, but often requiring assistance from other static analyses: abstract interpretation over intervals, and an adaptation of van Eijk’s

method for sequential equivalence checking. The experiments also show that strong versions of induction, such as k -induction with $k > 1$, are beneficial for equivalence proofs: several of our example proofs could only be conducted with $k \geq 2$.

Future work will involve completing the implementation of the van Eijk-based method by using random simulation to eliminate invalid pairs of signal variables, and analysing the effectiveness of our method over a larger class of Simulink designs.

References

1. Armoni, R., Fix, L., Fraer, R., Huddleston, S., Piterman, N., Vardi, M.Y.: SAT-based induction for temporal safety properties. *Electr. Notes Theor. Comput. Sci.* 119(2), 3–16 (2005)
2. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. *Advances in Computers* 58, 118–149 (2003)
3. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, W.R. (ed.) *TACAS 1999*. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999)
4. Bjese, P., Claessen, K.: SAT-based verification without state space traversal. In: Johnson, S.D., Hunt Jr., W.A. (eds.) *FMCAD 2000*. LNCS, vol. 1954, pp. 372–389. Springer, Heidelberg (2000)
5. Brillout, A., He, N., Mazzucchi, M., Kroening, D., Purandare, M., Rümmer, P., Weissenbacher, G.: Mutation-based test case generation for simulink models. In: de Boer, F.S., Bonsangue, M.M., Hallerstede, S., Leuschel, M. (eds.) *FMCO 2009*. LNCS, vol. 6286, pp. 208–227. Springer, Heidelberg (2010)
6. Clarke, E. M., Kröning, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) *TACAS 2004*. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004)
7. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: *Principles of Programming Languages (POPL)*, pp. 238–252. ACM, New York (1977)
8. Déharbe, D., Moreira, A.M.: Using induction and BDDs to model check invariants. In: *CHARME. IFIP Conference Proceedings*, vol. 105, pp. 203–213. Chapman & Hall, Boca Raton (1997)
9. DeMillo, R., Lipton, R., Sayward, F.: Hints on test data selection: Help for the practicing programmer. *Computer* 11(4), 34–41 (1978)
10. Donaldson, A.F., Haller, L., Kroening, D.: Strengthening induction-based race checking with lightweight static analysis. In: Jhala, R., Schmidt, D. (eds.) *VM-CAI 2011*. LNCS, vol. 6538, pp. 169–183. Springer, Heidelberg (2011)
11. Donaldson, A.F., Haller, L., Kroening, D., Rümmer, P.: Software verification using k -induction. In: Yahav, E. (ed.) *SAS 2011*. LNCS, vol. 6887, pp. 351–368. Springer, Heidelberg (2011)
12. Donaldson, A.F., Kroening, D., Rümmer, P.: Automatic analysis of scratch-pad memory code for heterogeneous multicore processors. In: Esparza, J., Majumdar, R. (eds.) *TACAS 2010*. LNCS, vol. 6015, pp. 280–295. Springer, Heidelberg (2010)
13. Donaldson, A.F., Kroening, D., Rümmer, P.: Automatic analysis of DMA races using model checking and k -induction. *Formal Methods in System Design* (2011)

14. Eén, N., Sörensson, N.: Temporal induction by incremental SAT solving. *Electr. Notes Theor. Comput. Sci.* 89(4) (2003)
15. van Eijk, C.A.J.: Sequential equivalence checking without state space traversal. In: *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, pp. 618–623. IEEE, Los Alamitos (1998)
16. Franzén, A.: Using satisfiability modulo theories for inductive verification of Lustre programs. *Electr. Notes Theor. Comput. Sci.* 144(1), 19–33 (2006)
17. Große, D., Le, H.M., Drechsler, R.: Proving transaction and system-level properties of untimed SystemC TLM designs. In: *MEMOCODE*, pp. 113–122. IEEE Computer Society, Los Alamitos (2010)
18. Hagen, G., Tinelli, C.: Scaling up the formal verification of Lustre programs with SMT-based techniques. In: *FMCAD*, pp. 109–117. IEEE, Los Alamitos (2008)
19. He, N., Rümmer, P., Kroening, D.: Test-case generation for embedded Simulink via formal concept analysis. In: *Proceedings of DAC* (2011)
20. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering, TSE* (2010)
21. Kuehlmann, A., van Eijk, C.A.J.: Combinational and sequential equivalence checking. In: *Logic Synthesis and Verification. Kluwer International Series in Engineering and Computer Science Series*, pp. 343–372. Kluwer, Dordrecht (2002)
22. Kupferman, O., Li, W., Seshia, S.A.: A theory of mutations with applications to vacuity, coverage, and fault tolerance. In: *Formal Methods in Computer-Aided Design (FMCAD)*, pp. 1–9. IEEE, Los Alamitos (2008)
23. Lillieroth, C.J., Singh, S.: Formal verification of FPGA cores. *Nord. J. Comput.* 6(3), 299–319 (1999)
24. Offutt, J., Voas, J.M.: Subsumption of condition coverage techniques by mutation testing. *Tech. Rep. ISSE-TR-96-01*, George Mason University (1996)
25. Ruthruff, J.R., Burnett, M.M., Rothermel, G.: Interactive fault localization techniques in a spreadsheet environment. *IEEE Transactions on Software Engineering (TSE)* 32(4), 213–239 (2006)
26. Sheeran, M., Singh, S., Stålmarck, G.: Checking safety properties using induction and a SAT-solver. In: Johnson, S.D., Hunt Jr., W.A. (eds.) *FMCAD 2000. LNCS*, vol. 1954, pp. 108–125. Springer, Heidelberg (2000)
27. Toom, A., Izerrouken, N., Naks, T., Pantel, M., Kai, O.S.Y.: Towards reliable code generation with an open tool: Evolutions of the Gene-Auto toolset. In: *Proceedings, Embedded Real Time Software and Systems, ERTS* (2010)
28. Vimjam, V.C., Hsiao, M.S.: Explicit safety property strengthening in SAT-based induction. In: *VLSID*, pp. 63–68. IEEE, Los Alamitos (2007)

The Hierarchical Compositional Interchange Format

Damian Nadales Agut, Bert van Beek, Harsh Beohar,
Pieter Cuijpers, and Jasper Fonteijn

Eindhoven University of Technology (TU/e)
The Netherlands

{D.E.Nadales.Agut,D.A.v.Beek,H.Beohar,P.J.L.Cuijpers}@tue.nl

Abstract. In computer science, the development of hierarchical automata / statecharts has lead to stepwise development of complex discrete systems. Such a concept is absent in the Compositional Interchange Format (CIF), which is a modelling language based on hybrid automata. In this article we extend the CIF language with the concept of hierarchy, which results in the Hierarchical Compositional Interchange format (HCIF). Syntactically, hierarchy is introduced by adding three concepts to CIF: a hierarchy function from a location to a HCIF composition, a termination predicate, and disruptive edges. The semantics of HCIF is given by means of Structural Operational Semantics rules. The semantics of a hierarchical automaton is defined in a compositional manner, by referring only to the transition system of the substructures, and not to their syntactic representation. This compositional introduction of hierarchy allows us to keep the semantics of the HCIF operators almost unchanged with respect to their CIF versions. Finally, a case-study called Patient Support System is modelled in HCIF to show its applicability¹.

1 Introduction

Hierarchy provides a structured and economical description of complex systems [21], which is suitable for incremental (bottom-up) construction of correct systems [3]. It also provides a framework for the development of abstraction and refinement techniques.

The Compositional Interchange Format (CIF) [5,4,2] is a language for modeling real-time, hybrid and embedded systems. CIF is developed to establish interoperability among a wide range of formalisms and associated tools for the specification of hybrid and timed systems, by means of model transformations to and from CIF. In this way, implementation of many bilateral translators, is avoided. As such it plays a central role in the European projects Multiform [16], HYCON [13], C4C [7], and HYCON 2 [12].

CIF has a formal semantics defined in terms of Structured Operational Semantics (SOS) [17] rules. This formal specification of the language is crucial for

¹ Work done as part of the European Community's Seventh Framework Programme (FP7/2007-2013) project MULTIFORM contract number FP7-ICT-224249.

enabling *semantic preserving* model transformations. Then, by translating CIF models to formalisms that have model checking algorithms for their models, such as Phaver [9], it is possible to verify CIF models as well.

In [6], the addition of hierarchy to a subset of CIF is investigated, and as a result it is shown that the SOS rules of atomic entities can be modified without altering the rules of the CIF operators. However the question remains whether this approach can be extended when more complicated concepts such as invariants [11], synchronization, and control variables [9] are added to the language.

In this paper we develop an extension of the full CIF language with hierarchy, named the Hierarchical Compositional Interchange Format (HCIF), and we model a case study in HCIF to show how the new concepts can be applied.

There exists several hierarchical formalisms and tools for simulation and validation of hybrid models, such as Charon [1], Matlab-Simulink [19], Statecharts [10], among others. However these formalisms either do not have a formal and compositional semantics, or their semantics is not defined in terms of SOS rules, which is a requirement for extending the CIF semantics.

The remainder of this work is organized as follows. In Section 2 the syntax of HCIF is introduced. In Section 3 we introduce the semantic framework needed to understand HCIF semantics, and in Section 4 we present the formal specification of the language. A case study that shows the applicability of the formalism is presented in Section 5.

2 Syntax of HCIF

In this section we describe the mathematical syntax of HCIF, and we illustrate it by modeling a controller of a simplified Patient Support System of an MRI scanner, which is discussed in more detail in Section 5. Note that in this section we give an incomplete description of the controller model to illustrate the various concepts involved in the definition of a hierarchical automaton.

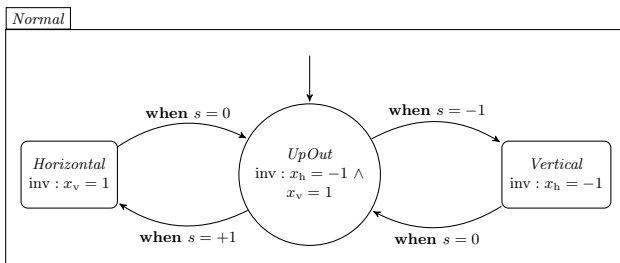


Fig. 1. Movement control

Fig. 1 gives an informal, graphical representation of a HCIF automaton, which models a controller of a patient support table. The control operates in one of the following three modes: *Horizontal* mode modeling the horizontal movement of the table, *UpOut* modeling that the table is fully up and out, and *Vertical* modeling the vertical movement of the table.

Every location has an initialization predicate, an *invariant* predicate, and a *time can progress predicate* associated to it. The initialization predicate of a location l describes the constraints that the initial values of variables must satisfy for an execution to start in l . Such locations for which the initialization predicate is true are called *active locations*. The *invariant* of a location l is a predicate that must hold as long as the system is in l . The *time can progress* (tcp) predicate of a location l is a predicate that must hold during time delays, when l is an active location. In Fig. 11, the location *UpOut* has true as the initialization predicate, $x_h = -1 \wedge x_v = 1$ as the invariant. Tcp predicates are in general useful for triggering the execution of an action from a location within a certain period of time. For instance, an action a must be executed when the clock value has reached 2 units of time (See Fig. 3(a)).

Edges represent discrete changes in the computational state of a system. An edge has a *source* and a *target* location, and its execution results in a change of active location (unless the edge is a self loop). The automaton of Fig. 11 has four edges in total among the locations *Horizontal*, *UpOut*, and *Vertical*. Every edge contains a predicate called *guard* that determines when an action can be executed, a predicate called *update* that determines how the model variables can change after performing the action, and a set of *jumping variables* that specify the variables that are changed by the action. Edges are labeled by *actions* that may be used to synchronize the behavior of automata in a parallel composition. In Fig. 11, the edge from the location *UpOut* to the location *Horizontal* has guard $s = 1$, update predicate true, empty set of jumping variables, and the silent action label τ .

Formally, the set of locations is denoted by \mathcal{L} and the set of actions is denoted by \mathcal{A} . The invisible action τ is a special symbol, which is not present in the set \mathcal{A} and we fix $\mathcal{A}_\tau = \mathcal{A} \cup \{\tau\}$. In HCIF there are three types of variables: regular variables, denoted by the set \mathcal{V} ; the dotted versions of those variables, which belong to the set $\dot{\mathcal{V}} = \{\dot{x} \mid x \in \mathcal{V}\}$; and the step variables, which belong to the set $\{x^+ \mid x \in \mathcal{V} \cup \dot{\mathcal{V}}\}$. The notation x^+ denotes the value of a variable x in the next state. Furthermore, the variables can be classified according to their evolution (i.e. how their values change during time delays). In particular, we distinguish between discrete variables (such as s in Fig. 11), whose values remain constant during time delays, so that the values of their dotted versions are always 0; and continuous variables (such as x_h and x_v in Fig. 11), whose values evolve as a continuous function of time during delays, and whose dotted versions represent their derivatives. Variables can also be constrained by differential algebraic equations, which are specified as predicates (in invariants). The values of the variables belong to the set \mathcal{A} that contains, among others, the sets \mathbb{B} (booleans) and \mathbb{R} (reals). The predicates representing the guards are taken from the set \mathcal{P}_g , the tcp, invariants and initializations are taken from the set \mathcal{P}_t and the resets are taken from the set \mathcal{P}_r . The exact syntax and semantics of predicates are defined in [2]. The predicates $\mathcal{P}_g, \mathcal{P}_t$ and \mathcal{P}_r are the terms of the language of predicate logic [18], where for $\mathcal{P}_g, \mathcal{P}_t$ the variables are taken from the set $\mathcal{V} \cup \dot{\mathcal{V}}$, and for \mathcal{P}_r the variables are taken from the set $\mathcal{V} \cup \dot{\mathcal{V}} \cup \{x^+ \mid x \in \mathcal{V} \cup \dot{\mathcal{V}}\}$.

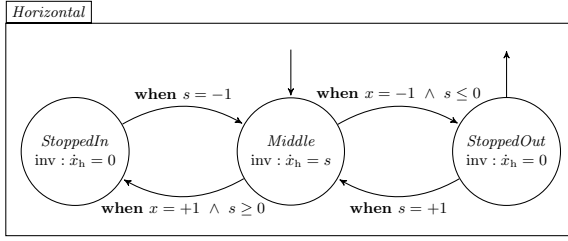


Fig. 2. Horizontal movement

Locations can contain other automata (or compositions of them, as we show in Section 5). In Fig. 1 the location *Horizontal* contains the automaton shown in Fig. 2, that defines the horizontal movement of the controller in more detail. Automata that are contained inside other locations are referred to as *sub-automata* or *sub-structure*, and the containing automata are referred to as *super-automata* or *super-structure*. In a HCIF automaton, there are two types of edges, namely, *non-disruptive* edges (for brevity, we refer to a non-disruptive edge as an edge) and *disruptive* edges. Intuitively, an edge can be executed from a location if the sub-structure at that location is terminating, while a disruptive edge can be executed even if the sub-structure at that location is non-terminating. Note that the conditions under which an edge or a disruptive edge can be executed depend on several factors, which are defined in Section 4.

In addition to initialization, invariant and tcp predicates, each location has a *termination predicate* which defines if execution can terminate in that location. Termination predicates are used for specifying when the super-structure can perform a transition. In the automaton shown in Fig. 1, the τ transition from the location *Horizontal* to the location *UpOut* can be executed only if the guard $s = 0$ holds, the automaton (Fig. 2) inside the location *Horizontal* has *StoppedOut* as its active location, and the termination predicate holds.

Additional components of an automaton (not shown in the example presented here) include: *control variables*, *synchronizing actions*, and *dynamic type* mappings. Intuitively, controlled variables are those variables that can only be modified by the automaton that declares them, and they do not change arbitrarily after performing an action. The set of synchronizing actions is used to specify which actions are to be synchronized when the automaton is composed in parallel. The concept of dynamic types [14] is used to model constraints in the joint evolution of a variable and its dotted version. In CIF a dynamic type is a set containing pairs of functions, whose domain is a closed range of the form $[0, t]$, with $t \in \mathbb{T}$. Notation \mathbb{T} is used to refer to the set of all time points.

Definition 1 (Hierarchical automata). A hierarchical automaton α is a tuple $(V, \text{init}, \text{inv}, \text{tcp}, E, D, \text{var}_C, \text{act}_S, \text{dtype}, \text{term}, h)$ where:

- $V \subseteq \mathcal{L}$ a set of locations,
- initial, invariant, time-can-progress and termination predicates $\text{init}, \text{inv}, \text{tcp}, \text{term}: V \rightarrow \mathcal{P}_t$,

- a set of edges $E \subseteq V \times \mathcal{P}_g \times \mathcal{A} \times (2^{\mathcal{V} \cup \mathcal{V}} \times \mathcal{P}_r) \times V$,
- $D \subseteq E$ is the set of disruptive edges of the automaton,
- $\text{var}_C \subseteq 2^{\mathcal{V}}$ is the set of controlled variables,
- $\text{act}_S \subseteq 2^{\mathcal{A}}$ is the set of synchronizing actions, and
- $\text{dtype} : \mathcal{V} \rightarrow 2^{(\mathbb{T} \rightarrow \mathcal{A}) \times (\mathbb{T} \rightarrow \mathcal{A})}$ is the dynamic type mapping.
- $h : V \rightarrow \mathcal{C}$ is a partial function that associates to some set of locations a sub-structure. Here, \mathcal{C} is the set of all compositions in HCIF (See Definition 2).

We use symbol \mathcal{M} to refer to the set of all hierarchical automata.

Using operators, more complex models, referred to as compositions (Definition 2), are possible. The semantics of the operators is presented in Section 4.1, with the exception of the semantics of the action and variable scope operators. The semantics of these operators is unchanged with respect to the semantics of these operators in CIF, as defined in 2.

Definition 2 (HCIF compositions). *The set of compositions \mathcal{C} in the HCIF formalism is recursively defined by the grammar below, where $x \in \mathcal{V}$, $e \in \mathcal{E}$, $a \in \mathcal{A}$, $a_\tau \in \mathcal{A}_\tau$. Informally, by a composition we mean either a hierarchical automaton or a syntactical object constructed from the different hierarchical automata using the operators of HCIF. Note that, the word ‘composition’ is synonymous to the phrase ‘process term’ used in process algebra terminology.*

$\mathcal{C} ::= \alpha$	hierarchical automaton
$\mathcal{C} : \alpha$	automaton postfix operator
$\mathcal{C} \parallel \mathcal{C}$	parallel composition operator
$[[\mathcal{V} \ x = e, \dot{x} = e :: \mathcal{C}]]$	variable scope operator
$[[\mathcal{A} \ a :: \mathcal{C}]]$	action scope operator
$\mathbf{v}_{a_\tau}(\mathcal{C})$	urgency operator

Throughout this article, the textual and graphical conventions given in Table 1 are followed.






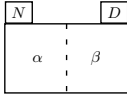
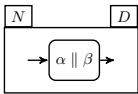
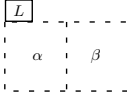
3 Semantic Framework

In this section, the semantic framework is set up to properly explain the semantics of HCIF. First we present the concepts of variable valuations and flow trajectories. Next we describe informally hybrid transitions systems, which are used to model the semantics of HCIF compositions. Finally, a formal definition of this semantic model is given.

3.1 Preliminaries

Semantically, the *execution* of a system, specified by means of a HCIF composition, causes changes to the values of the variables appearing on it. Thus, in the semantic model it is necessary to represent the values of the variables in a

Table 1. Textual and graphical conventions in HCIF. The sentences terminated with the symbol \star indicates the features present only in HCIF.

Graphical representation	Meaning
	Location without any sub-structure
	Initial location with init predicate true
	Final location with termination predicate true \star
	Location containing a sub-structure \star
when g act a do $x := e$	Edge $(g, a, (\{x\}, x^+ = e))$
when g act a do $x := e$	Disruptive edge $(g, a, (\{x\}, x^+ = e))\star$
when g	Edge $(g, \tau, (\emptyset, \text{true}))$
act a	Edge $(\text{true}, a, (\emptyset, \text{true}))$
	Automaton N with declarations D
	
	AND superstate L containing parallel composition $\alpha \parallel \beta$

particular instant. For this purpose, we use the concept of *valuation*, which is standard in semantics of processes with data. A valuation $\sigma: (\mathcal{V} \cup \dot{\mathcal{V}}) \rightarrow \mathcal{A}$ is a function that for each variable returns its corresponding value. We use notation $\Sigma \triangleq (\mathcal{V} \cup \dot{\mathcal{V}}) \rightarrow \mathcal{A}$ to refer to the set of all valuations.

Having defined valuations, we introduce the concept of satisfiability. Even though predicates are abstract entities, we assume that a satisfaction relation $\sigma \models u$ is defined, which expresses that predicate $u \in \mathcal{P}$ is satisfied (i.e. it is true) in valuation σ . For a valuation σ , we define $\sigma^+ \triangleq \{(v^+, c) \mid (v, c) \in \sigma\}$.

To model the evolution on the values of variables during time delays we use the concept of *variable trajectories*. A variable trajectory is a function $\rho: \mathbb{T} \rightarrow \Sigma$ that returns the valuations of the variables at each time point. In other words, $\rho(s)(x)$ is the value of variable x at time s . We assume the domain of variable trajectories to be closed intervals, i.e. intervals of the form $[0, t]$, where $t \in \mathbb{T}$.

3.2 Hybrid Transition Systems

The semantics of CIF compositions is given in terms of SOS rules, which induce hybrid transition systems (HTS) [8]. The states of the HTS are of the form $\langle p, \sigma \rangle$, where $p \in \mathcal{C}$ and $\sigma \in \Sigma$ is a valuation. There are three kind of transition in the HTS, namely, *action transitions*, *environment transitions*, and *time transitions*.

Action transitions are of the form $\langle p, \sigma \rangle \xrightarrow{a,b,X} \langle p', \sigma' \rangle$. They model the execution of action a by process p in an initial valuation σ , which changes process p into p' and results in a valuation σ' . Label b is a boolean that indicates whether action a is synchronizing or not, and label X is the set of controlled variables defined by the environment of p and p' .

Time behavior is captured by *time transitions*. Time transitions are of the form $\langle p, \sigma \rangle \xrightarrow{\rho, A, \theta, \omega} \langle p', \sigma' \rangle$. They model the passage of time in composition p , in an initial valuation σ , which results in a composition p' and valuation σ' . Label A contains the set of synchronizing actions of p and p' . Function $\rho : \mathbb{T} \rightarrow \Sigma$ is the variable trajectory. Function $\theta : \mathbb{T} \rightarrow 2^A$ is called *guard trajectory*. It models the evolution of enabled actions during time delays. For each time point $s \in \text{dom}(\theta)$, the function application $\theta(s)$ yields the set of enabled actions of composition p at time s . Lastly, function ω is called *termination trajectory*. It models the evolution of termination (see below) during time delays: for each time point $s \in \text{dom}(\omega)$, composition p' is terminating at time s if and only if $\omega(s)$. For all time transition $\text{dom}(\rho) = [0, t]$, for some time point $t \in \mathbb{T}$, and $\text{dom}(\rho) = \text{dom}(\theta) = \text{dom}(\omega)$. Termination is formally defined next.

Definition 3. *Given a valuation σ , we define termination as follows:*

- An automaton $(V, \text{init}, \text{inv}, \text{tcp}, E, \text{var}_C, \text{act}_S, \text{dtype}, \text{term}, \text{h})$ is terminating in σ if there is a location $v \in V$ such that $\sigma \models \text{init}(v)$, $\sigma \models \text{inv}(v)$, $\sigma \models \text{term}(v)$, and if $v \in \text{dom}(\text{h})$ then $\text{h}(v)$ is terminating in σ .
- Composition $p \parallel q$ is terminating in valuation σ if p and q are terminating in valuation σ .
- For the remaining operators, termination is defined pointwise.

Environment transitions are of the form $\langle p, \sigma \rangle \xrightarrow{A,b} \langle p', \sigma' \rangle$. They are used in the semantics to enforce restrictions posed by the environment of a composition on the action behavior of the composition. More specifically, a transition $\langle p, \sigma \rangle \xrightarrow{A,b} \langle p', \sigma' \rangle$ expresses the fact that p is consistent in σ , and p' is consistent in σ' . In addition, the role of the environment transitions is to indicate that a composition p can initialize to become a composition p' in which an active location is fixed for each (active) substructure. Furthermore, the boolean b indicates whether the initialized substructure can terminate, and thus give back the control over actions to its environment. As before, label A is the set of synchronizing actions of compositions p and p' . Next, *consistency* is defined recursively.

Definition 4. *Given a valuation σ , we define consistency as follows.*

- An automaton $(V, \text{init}, \text{inv}, \text{tcp}, E, \text{var}_C, \text{act}_S, \text{dtype}, \text{term}, \text{h})$ is consistent in σ if there is a location $\ell \in V$ such that $\sigma \models \text{init}(\ell)$ and $\sigma \models \text{inv}(\ell)$, and if $\ell \in \text{dom}(\text{h})$ then $\text{h}(\ell)$ is consistent in σ .
- Composition $p \parallel q$ is consistent in valuation σ if p and q are consistent in valuation σ .
- For the remaining operators, consistency is defined pointwise.

We use notation $\sigma \models p$ to denote that composition p is consistent in valuation σ . Alternatively, we say that σ is consistent with p .

Definition 5 formalizes the hybrid transition system induced by the SOS rules presented in the next sections.

Definition 5. A hybrid transition system (HTS) is a five-tuple of the form $(Q, \mathcal{A}, \rightarrow, \mapsto, \dashrightarrow)$ where $Q \triangleq \mathcal{C} \times \Sigma$, $\rightarrow \subseteq Q \times (\mathcal{A}_\tau \times \mathbb{B} \times 2^V) \times Q$, $\mapsto \subseteq Q \times ((\mathbb{T} \rightarrow \Sigma) \times 2^A \times (\mathbb{T} \rightarrow 2^A) \times (\mathbb{T} \rightarrow \mathbb{B})) \times Q$, $\dashrightarrow \subseteq Q \times (2^A \times \mathbb{B}) \times Q$.

4 Semantics

In this section we explain the semantics of HCIF both informally by means of examples, and formally by means of SOS rules.

4.1 Hierarchical Automata

In a hierarchical automaton α , an active location v can execute actions at two different levels of abstraction: *external* actions, which are specified as labelled edges from the active location v to an arbitrary location v' ; and *internal* actions, which are generated by the sub-structure at the location v , i.e., $h(v)$. Note that there are different conditions under which an external or internal action can be executed. Furthermore, the rules of CIF can be obtained from the current rules by substituting $h = \emptyset$. Next, we explain and formalize the rules for every HCIF composition.

Given an initial valuation σ , an external action a can be executed in a location v if there is an edge $(v, g, a, (W, r), v')$ in α satisfying the following conditions:

- Location v is active ($\sigma \models \text{init}(v)$), the invariant at the location v is satisfied ($\sigma \models \text{inv}(v)$) and the guard g holds ($\sigma \models g$).
- If there is a substructure inside location v ($v \in \text{dom}(h)$), then it is terminating in σ or the edge is disruptive.
- It is possible to find a new valuation σ' such that:
 - The invariant of the new location v' holds $\sigma \models \text{inv}(v')$.
 - The reset predicate r is satisfied in valuation $\sigma \cup \sigma'^+$ ($\sigma \cup \sigma'^+ \models r$).
 - σ' is consistent with the substructure inside the target location (if any).
 - Controlled variables not in W (the set of jumping variables of the action) are not allowed to change in σ' (wrt. σ).

This is formalized by Rule II. Some of the above conditions are summarized in the term

$$\sigma, \sigma' \models_\alpha (v, g, a, (W, r), v')$$

that is syntactically equivalent to:

$$(v, g, a, r, v') \in E \wedge \sigma \models \text{init}(v) \wedge \sigma \models g \wedge \sigma \models \text{inv}(v) \wedge \sigma' \models \text{inv}(v') \wedge \sigma'^+ \cup \sigma \models r.$$

Henceforth, we use the following notation:

$$\begin{aligned}\alpha &\equiv (V, \text{init}, \text{inv}, \text{tcp}, E, \text{var}_C, \text{act}_S, \text{dtype}, \text{term}, h), \\ \alpha[v] &\equiv (V, \text{id}_v, \text{inv}, \text{tcp}, E, \text{var}_C, \text{act}_S, \text{dtype}, \text{term}, h),\end{aligned}$$

where $\text{dom}(\text{id}_v) = V$ and $\text{id}_v(w) \triangleq v = w$. Note that the initialization predicate id_v encodes the active location, after execution of the first transition. An action specified by an edge $(v, g, a, (W, r), v')$ in an automaton α can be triggered only if the controlled variables of the automaton (var_C) and of the environment (X) remain the same in σ , and σ' , except if they belong to the set of jumping variables W . We use notation $f \upharpoonright_A$ to refer to the domain restriction of function f to the set A . Secondly, if the edge is not disruptive, it is necessary to check that the substructure of the initial location, if any, is terminating. This is expressed by condition $(\langle h(v), \sigma \rangle \xrightarrow{A_0, b} \langle p, \sigma \rangle \vee v \notin \text{dom}(h)), (v, g, a, (W, r), v') \in D \vee b$. Finally, after the action is performed, the substructure in the target location, if present, must be initialized, i.e., $\langle h(v'), \sigma' \rangle \xrightarrow{A_1, b} \langle q : \alpha[v'], \sigma' \rangle$. The choice of selecting active locations of substructure $h(v')$ is made upon entering the location v' . Consistency of the substructures is preserved by the environment transitions.

$$\frac{\begin{array}{l} \sigma, \sigma' \models_{\alpha} (v, g, a, (W, r), v'), \quad \sigma \upharpoonright_{(X \cup \text{var}_C) \setminus W} = \sigma' \upharpoonright_{(X \cup \text{var}_C) \setminus W}, \\ (\langle h(v), \sigma \rangle \xrightarrow{A_0, b} \langle p, \sigma \rangle \vee v \notin \text{dom}(h)), \quad (v, g, a, (W, r), v') \in D \vee b, \\ v' \in \text{dom}(h), \quad \langle h(v'), \sigma' \rangle \xrightarrow{A_1, b'} \langle q, \sigma' \rangle \end{array}}{\langle \alpha, \sigma \rangle \xrightarrow{a, a \in \text{act}_S, X} \langle q : \alpha[v'], \sigma' \rangle} 1$$

Consider the controller automaton in Fig. 1, assuming *UpOut* is an active location with a valuation σ . The edge labelled **when** $s = +1$ can be executed if there exists a valuation σ' such that σ satisfies the invariant of the location *UpOut* ($\sigma(x_h) = -1 \wedge \sigma(x_v) = 1$), σ' satisfies the invariant of the location *Horizontal* ($\sigma'(x_v) = 1$) and the valuation σ' is consistent with the automaton shown in Fig. 2. The consistency of the valuation σ' implies that the active location of the automaton in Fig. 2 is *Middle* such that $\sigma' \models \dot{x}_h = s$.

Now consider the active location of the controller to be *Horizontal* and the active location of the automaton in Fig. 2 to be *Stopped-in*. In this case, the edge labelled **when** $s = 0$ in Fig. 1 cannot be executed even if the guard $s = 0$ is true. This is due to the fact that the composition inside the location *Horizontal* is non-terminating in location *Stopped-in*. External actions, such as the edge labelled **when** $s = 0$, can be executed only if either the sub-structure is terminating or the edge labelled with the external action is specified as disruptive.

Rule 1 requires as a condition that there is an active substructure in the target location $v' \in \text{dom}(h)$. If this is not the case then no active substructure is prefixed to $\alpha[v]$, as expressed by Rule 2.

$$\frac{\begin{array}{l} \sigma, \sigma' \models_{\alpha} (v, g, a, (W, r), v'), \quad \sigma \upharpoonright_{(X \cup \text{var}_C) \setminus W} = \sigma' \upharpoonright_{(X \cup \text{var}_C) \setminus W}, \\ (\langle h(v), \sigma \rangle \xrightarrow{A_0, b} \langle p, \sigma \rangle \vee v \notin \text{dom}(h)), \quad v' \notin \text{dom}(h), \quad (v, g, a, (W, r), v') \in D \vee b \end{array}}{\langle \alpha, \sigma \rangle \xrightarrow{a, a \in \text{act}_S, X} \langle \alpha[v'], \sigma' \rangle} 2$$

Besides external actions, an automaton can also execute internal actions, which are triggered by their internal structures. Given a valuation σ , an internal action can be executed in an initial location v if the following conditions hold:

- The invariant associated with location v is satisfied ($\sigma \models \text{inv}(v)$).
- The action performed by the substructure of v results in a new valuation σ' that also satisfies the invariant of v .

Rule [3](#) formalizes this. In the conclusion, $p : \alpha[v]$ reflects the fact that an initial location is chosen in a hierarchical structure if the substructure performs an action. In this rule we ignore the boolean b , that indicates whether the action a is synchronizing, in the premise. As a result, the superstructure decides on which actions it wants to synchronize. In other words, the superstructure defines the set of synchronizing actions, independently of the sub-levels.

$$\frac{\sigma \models \text{init}(v), \sigma \models \text{inv}(v), \sigma' \models \text{inv}(v), v \in \text{dom}(h), \langle h(v), \sigma \rangle \xrightarrow{a, b, X \cup \text{var}_C} \langle p, \sigma' \rangle}{\langle \alpha, \sigma \rangle \xrightarrow{a, a \in \text{act}_S, X} \langle p : \alpha[v], \sigma' \rangle} \quad 3$$

Transition $\langle h(v), \sigma \rangle \xrightarrow{a, b, X \cup \text{var}_C} \langle p, \sigma' \rangle$ in the premise of the above rule ensures that the control variables inherited from the environment (X) as well as the control variables of the automaton (var_C) will not jump arbitrarily when the action is carried out by the substructure.

Again consider the model of the controller as given in Figures [1](#) and [2](#). Assume that the active location is *Horizontal* and the active location of the substructure is *Middle*. The edge labelled **when** $x \geq 1 \wedge s \geq 0$ can be executed from the location *Middle* only if there exists a new valuation σ' such that it satisfies the invariant of the locations *Horizontal* and *StoppedIn*.

In hierarchical CIF, a time delay is possible in an active location v if there exists a trajectory ρ such that the invariant associated with the active locations is satisfied in time point $[0, t]$, the tcp predicate is satisfied in $[0, t)$ and the dynamic type constraints specified by dtype are satisfied. Henceforth, we use $\rho \models \langle t, v, \text{init}, \text{inv}, \text{tcp}, \text{dtype} \rangle$ as an abbreviation of the predicate

$$\begin{aligned} \rho(0) \models \text{init}(v) \wedge \text{dom}(\rho) = [0, t] \wedge 0 < t \wedge \forall_{s \in [0, t)} \cdot \rho(s) \models \text{tcp}(v) \wedge \\ \forall_{s \in [0, t]} \cdot \rho(s) \models \text{inv}(v) \wedge \forall_{x \in \text{dom}(\text{dtype})} \cdot (\rho \downarrow x, \rho \downarrow \dot{x}) \in \text{dtype}(x). \end{aligned}$$

For time delays, the substructure (if present) must perform a time transition with the same trajectory. In this way, the invariants and tcp-predicates of the active location of the automaton, and, recursively, of the active locations of its active substructures are considered simultaneously. In this way time passes in an automaton, and also in all of its contained active substructures. In other words, an automaton and its active substructure synchronize on time delays. Rule [4](#) models this, where $\text{dom}(\omega) = \text{dom}(\rho)$, $\text{dom}(\theta) = \text{dom}(\rho)$, $\forall_{s \in [0, t]} \cdot \omega(s) = (\omega_0(s) \wedge \rho(s) \models \text{term}(v))$, and $\forall_{s \in [0, t]} \cdot \theta(s) = \theta_0(s) \cup \{a \mid (v, g, a, (W, r), v') \in E \wedge \rho(s) \models g \wedge \omega_0(s)\}$. The guard trajectory θ as well as the termination trajectory ω are constructed by using the corresponding trajectories generated by the time

transition in the substructure. The set of synchronizing actions only takes into account the set act_S in the superstructure, since the set of synchronizing actions in the substructure does not influence the action synchronizing behavior of its parent. The same approach is taken when computing the set of synchronizing actions in the environment transition in Rule [6](#).

$$\frac{\rho \models \langle t, v, \text{init}, \text{inv}, \text{tcp}, \text{dtype} \rangle, \quad v \in \text{dom}(h), \quad \langle h(v), \rho(0) \rangle \xrightarrow{\rho, A, \theta_0, \omega_0} \langle p, \rho(t) \rangle}{\langle \alpha, \rho(0) \rangle \xrightarrow{\rho, \text{act}_S, \theta, \omega} \langle p' : \alpha[v], \rho(t) \rangle} 4$$

Rule [5](#) deals with the case that an initial location v does not contain a substructure, where $\text{dom}(\omega) = \text{dom}(\rho)$, $\text{dom}(\theta) = \text{dom}(\rho)$ and $\forall_{s \in [0, t].} \omega(s) = (\rho(s) \models \text{term}(v))$, and $\forall_{s \in [0, t].} \theta(s) = \{a \mid (v, g, a, (W, r), v') \in E \wedge \rho(s) \models g\}$.

$$\frac{\rho \models \langle t, v, \text{init}, \text{inv}, \text{tcp}, \text{dtype} \rangle, \quad v \notin \text{dom}(h)}{\langle \alpha, \rho(0) \rangle \xrightarrow{\rho, \text{act}_S, \theta, \omega} \langle \alpha[v], \rho(t) \rangle} 5$$

In hierarchical CIF, if an automaton performs an environment transition, a unique active location is chosen, and the substructure (if present) is initialized. The environment transition ensures that the active location contains a consistent hierarchical structure (Definition [4](#)). This is expressed by Rule [6](#). The initialized composition p becomes the active substructure of $\alpha[v]$, and the automaton is terminating if the location and the active substructure are. Rule [7](#) deals with the case where there is no substructure.

$$\frac{\begin{array}{l} \sigma \models \text{init}(v), \quad \sigma \models \text{inv}(v), \quad \sigma' \models \text{inv}(v), \quad \sigma \upharpoonright_{\text{var}_C} = \sigma' \upharpoonright_{\text{var}_C}, \\ v \in \text{dom}(h), \quad \langle h(v), \sigma \rangle \xrightarrow{A, b} \langle p', \sigma' \rangle \end{array}}{\langle \alpha, \sigma \rangle \xrightarrow{\text{act}_S, \sigma \models \text{term}(v) \wedge b} \langle p' : \alpha[v], \sigma' \rangle} 6$$

$$\frac{\sigma \models \text{init}(v), \quad \sigma \models \text{inv}(v), \quad \sigma' \models \text{inv}(v), \quad \sigma \upharpoonright_{\text{var}_C} = \sigma' \upharpoonright_{\text{var}_C}, \quad v \notin \text{dom}(h)}{\langle \alpha, \sigma \rangle \xrightarrow{\text{act}_S, \sigma \models \text{term}(v)} \langle \alpha[v], \sigma' \rangle} 7$$

4.2 Automaton Postfix Operator

The automaton postfix operator is used to define the semantics of hierarchy. It is not an operator intended for modeling, and therefore we do not illustrate its behavior by means of examples. We limit ourselves to semantic considerations.

Intuitively, the composition $p : \alpha$ means that composition p is the active substructure of some initial location $v \in V$ in the automaton α . Note, that whenever the composition $p : \alpha$ is the result of a previous transition in α , this initial location is always uniquely defined.

Rule [8](#) models the action transition taken by automaton α when the active substructure is terminating or when the chosen edge is disruptive, and the target location has a substructure. Rule [9](#) differs from Rule [8](#) only in that the target location does not have a substructure. Rule [10](#) models the action transition resulting from the execution of the substructure.

$$\frac{\sigma, \sigma' \models_{\alpha} (v, g, a, (W, r), v'), \quad \sigma \upharpoonright_{(X \cup \text{var}_C) \setminus W} = \sigma' \upharpoonright_{(X \cup \text{var}_C) \setminus W}, \quad \langle p, \sigma \rangle \xrightarrow{A_0, b} \langle p', \sigma \rangle,}{\langle p : \alpha, \sigma \rangle \xrightarrow{a, a \in \text{acts}_S, X} \langle q : \alpha[v'], \sigma' \rangle} \quad 8$$

$$\frac{\sigma, \sigma' \models_{\alpha} (v, g, a, (W, r), v'), \quad \sigma \upharpoonright_{(X \cup \text{var}_C) \setminus W} = \sigma' \upharpoonright_{(X \cup \text{var}_C) \setminus W}, \quad \langle p, \sigma \rangle \xrightarrow{A, b} \langle p', \sigma \rangle, \quad (v, g, a, (W, r), v') \in D \vee b, \quad v' \notin \text{dom}(h)}{\langle p : \alpha, \sigma \rangle \xrightarrow{a, a \in \text{acts}_S, X} \langle \alpha[v'], \sigma' \rangle} \quad 9$$

$$\frac{\sigma \models \text{init}(v), \quad \sigma \models \text{inv}(v), \quad \sigma' \models \text{inv}(v), \quad \langle p, \sigma \rangle \xrightarrow{a, b, X \cup \text{var}_C} \langle q, \sigma' \rangle}{\langle p : \alpha, \sigma \rangle \xrightarrow{a, a \in \text{acts}_S, X} \langle q : \alpha, \sigma' \rangle} \quad 10$$

Rule [11](#) models the passage of time in an automaton postfix such that the timed transitions are (recursively) synchronized in every level of hierarchy of $p : \alpha$, where $\text{dom}(\omega) = \text{dom}(\rho)$, $\text{dom}(\theta) = \text{dom}(\rho)$, $\forall_{s \in [0, t]} \omega(s) = \omega_0(s) \wedge \rho(s) \models \text{term}(v)$, and $\forall_{s \in [0, t]} \theta(s) = \theta_0(s) \cup \{a \mid (v, g, a, (W, r), v') \in E \wedge \rho(s) \models g \wedge \omega_0(s)\}$.

$$\frac{\rho \models \langle t, v, \text{init}, \text{inv}, \text{tcp}, \text{dtype} \rangle, \quad \langle p, \rho(0) \rangle \xrightarrow{\rho, A, \theta_0, \omega_0} \langle p', \rho(t) \rangle}{\langle p : \alpha, \rho(0) \rangle \xrightarrow{\rho, \text{acts}_S, \theta, \omega} \langle p' : \alpha[v], \rho(t) \rangle} \quad 11$$

Finally, Rule [12](#) models the execution of an environment transition in an automaton postfix.

$$\frac{\sigma \models \text{init}(v), \quad \sigma \models \text{inv}(v), \quad \sigma' \models \text{inv}(v), \quad \sigma \upharpoonright_{\text{var}_C} = \sigma' \upharpoonright_{\text{var}_C}, \quad \langle p, \sigma \rangle \xrightarrow{A, b} \langle p', \sigma' \rangle}{\langle p : \alpha, \sigma \rangle \xrightarrow{\text{acts}_S, \sigma \models \text{term}(v) \wedge b} \langle p' : \alpha[v], \sigma' \rangle} \quad 12$$

4.3 Parallel Composition

The parallel composition operator allows concurrent execution of HCIF compositions. The semantics of parallel composition is equal to the CIF semantics. Action behavior is not affected by the addition of hierarchy. The rules for time and environment transitions are updated to reflect the fact that a parallel composition is terminating only if both components are.

As an illustration, consider the assembly process shown in Fig. [3\(a\)](#), henceforth referred to as *Assembly*, such that its location *WaitForAB* contains the parallel composition shown in Fig. [3\(b\)](#). The assembly process initially is in the *WaitForAB* location, and, according to the semantics of atomic automata, it can trigger action *assembling* only if its sub-structure terminates. Since the sub-structure is a parallel composition of two automata, namely *WaitForA* and *WaitForB* (See Fig. [3\(b\)](#)), the substructure $h(\text{WaitForAB})$ can terminate after actions a and b have both been executed; i.e., both automata *WaitForA* and

WaitForB can terminate. This pattern, in which an action is triggered after a series of parallel processes terminate, can be expressed succinctly using hierarchy. Without support for hierarchy and termination it is necessary to rewrite the parallel processes into a flat automaton.

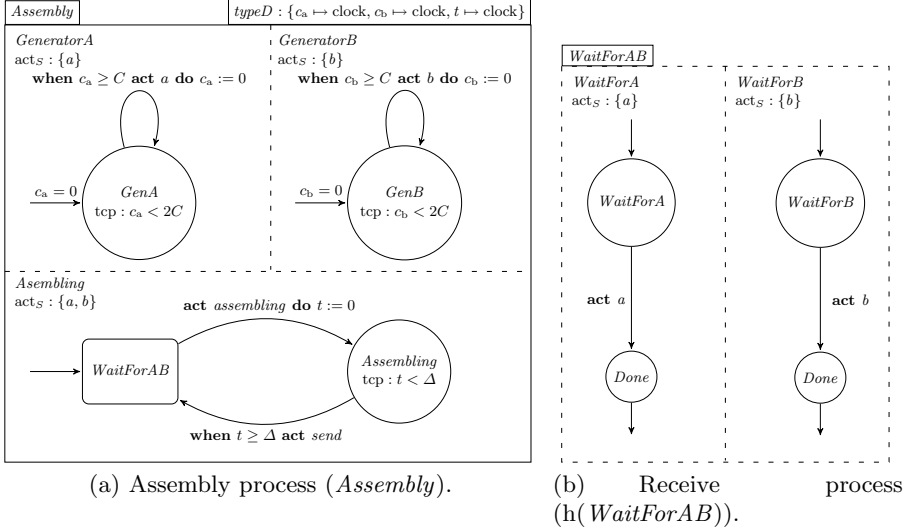


Fig. 3. Assembly line

The addition of hierarchy facilitates inter-level synchronization. As an example consider the generator process *GeneratorA* shown in Fig. 3(a), which enables an action *a* every *C* time units, when $c_a \geq C$. The action *a* from the generator synchronizes with action *a* specified as synchronizing in the automaton *WaitForA*, which is part of the substructure of location *WaitForAB*. This synchronizing behavior is obtained by inclusion of action *a* in the set of synchronizing actions $acts_S$ of *GeneratorA* ($\{a\}$), and in the set of synchronizing actions of automaton *Assembly* ($\{a, b\}$). Note that strictly speaking, action *a* need not be defined as synchronizing for automaton *WaitForA*.

Formally, Rule 13 states that two synchronizing actions with the same label can execute in parallel only if they share the same initial and final valuation, and if the action is synchronizing in both the compositions. The set of control variables *X*, is propagated from the conclusions to the premises since the control variables in the scope of a parallel composition are shared by both partners. The resulting action transition is also synchronizing which allows action *a* to synchronise with more than two compositions.

$$\frac{\langle p, \sigma \rangle \xrightarrow{a, \text{true}, X} \langle p', \sigma' \rangle, \quad \langle q, \sigma \rangle \xrightarrow{a, \text{true}, X} \langle q', \sigma' \rangle}{\langle p \parallel q, \sigma \rangle \xrightarrow{a, \text{true}, X} \langle p' \parallel q', \sigma' \rangle} \quad 13$$

Rules **I4** model interleaving behavior of two compositions when executed in parallel. In these rules, an action can be performed in one of the components (p) only if the initial and final valuations are consistent with the other composition (q); and if this action is not synchronizing in the other component, which is expressed by the condition $a \notin A$. The environment transition $(q, \sigma) \xrightarrow{A, b'} (q', \sigma')$ is used to obtain the set of synchronizing action labels in composition q , to ensure that the initial valuation σ is consistent with the active invariants and initialization conditions of q , to select an initial location (in case there is more than one in q), and to remove any initialization operators from q .

$$\frac{\langle p, \sigma \rangle \xrightarrow{a, b, X} \langle p', \sigma' \rangle, \quad \langle q, \sigma \rangle \xrightarrow{A, b'} \langle q', \sigma' \rangle, \quad a \notin A}{\langle p \parallel q, \sigma \rangle \xrightarrow{a, b, X} \langle p' \parallel q', \sigma' \rangle} \quad 14$$

$$\langle q \parallel p, \sigma \rangle \xrightarrow{a, b, X} \langle q' \parallel p', \sigma' \rangle$$

Rule **I5** models the fact that if two compositions are put in parallel, time can pass t time units only if allowed by both partners. As can be seen in this rule, the set of enabled actions in the parallel composition at any point in time during the delay depends both on the set of enabled actions and the set of synchronizing actions in each component individually. Similarly, the termination trajectory of the parallel composition depends on the termination trajectories of its components, where $\theta_{01} = (\theta_0 \cap \theta_1) \cup (\theta_0 \setminus A_1) \cup (\theta_1 \setminus A_0)$ and $\forall s \in [0, t]. [\omega_{01}(s) = \omega_0(s) \wedge \omega_1(s)]$.

$$\frac{\langle p, \rho(0) \rangle \xrightarrow{\rho, A_0, \theta_0, \omega_0} \langle p', \rho(t) \rangle, \quad \langle q, \rho(0) \rangle \xrightarrow{\rho, A_1, \theta_1, \omega_1} \langle q', \rho(t) \rangle}{\langle p \parallel q, \rho(0) \rangle \xrightarrow{\rho, A_0 \cup A_1, \theta_{01}, \omega_{01}} \langle p' \parallel q', \rho(t) \rangle} \quad 15$$

Rule **I6** defines the environment transition behavior for parallel composition. The resulting set of synchronizing actions is the union of the synchronizing actions of p and q . The conjunction $b_0 \wedge b_1$ models the fact that a parallel composition is terminating if its components are. Note that the end valuations of all transitions match.

$$\frac{\langle p, \sigma \rangle \xrightarrow{A_0, b_0} \langle p', \sigma' \rangle, \quad \langle q, \sigma \rangle \xrightarrow{A_1, b_1} \langle q', \sigma' \rangle}{\langle p \parallel q, \sigma \rangle \xrightarrow{A_0 \cup A_1, b_0 \wedge b_1} \langle p' \parallel q', \sigma' \rangle} \quad 16$$

4.4 Urgency Operator

By means of the urgency operator it is possible to declare actions as urgent. This means that time cannot pass if an urgent action is enabled. However, urgent actions do not have priority over regular (non-urgent) actions.

For example, consider the model of the controller of Fig. **I** with the active location $UpOut$. When a user demands the controller to operate in the horizontal mode, it should react as soon as possible. In other words, the action τ in the

labelled edge **when** $s = +1$ between the locations *UpOut* and *Horizontal* must be made urgent. This ensures that time does not pass in the location *UpOut* from the instant when the guard $s = +1$ is enabled.

Rule 17 specifies that the urgent action operator allows the passage of time as long as no urgent action is enabled.

$$\frac{(p, \sigma) \xrightarrow{\rho, A, \theta, \omega} (p', \sigma'), \quad \forall_{s \in [0, t]} \cdot a \notin \theta(s)}{(v_a(p), \sigma) \xrightarrow{\rho, A, \theta, \omega} (v_a(p'), \sigma')} \quad 17$$

The urgency operator affects only the time behavior. Action and environment transitions remain unchanged as expressed by Rules 18 and 19.

$$\frac{(p, \sigma) \xrightarrow{\ell, b, X} (p', \sigma')}{(v_a(p), \sigma) \xrightarrow{\ell, b, X} (v_a(p'), \sigma')} \quad 18 \qquad \frac{(p, \sigma) \xrightarrow{A, b} (p', \sigma')}{(v_a(p), \sigma) \xrightarrow{A, b} (v_a(p'), \sigma')} \quad 19$$

5 Case-Study: Patient Support System

The patient support system (See Fig. 4) is used in medical diagnosis to position a patient in an MRI scanner [20]. The system can be operated in the following modes: vertical mode, horizontal mode and user interface mode. In the vertical mode, the table top on which a patient resides can only move vertically between the bounds depicted in Fig. 4. Similarly, in the horizontal mode, the table top can be moved in or out of the bore, either manually or by means of a motor drive. Furthermore, the system is equipped with a table top release switch for emergency situations. This system is controlled via a user interface that contains a tumble switch to control the movement (both horizontally and vertically) of the table, and a button to enable the start of an initialization sequence. The position of the tumble switch is represented by variable s which can have the values $+1, 0$ and -1 . The continuous variables x_h and x_v represent the horizontal and vertical position of the table top, respectively.

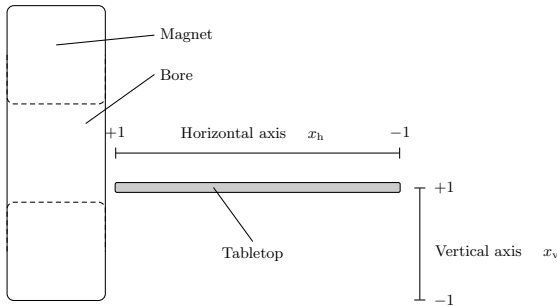


Fig. 4. Patient Support System

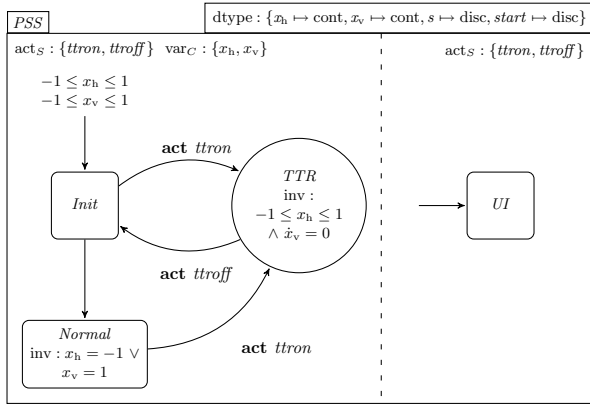


Fig. 5. Patient Support System

The objective of this case-study is to design a controller that satisfies the following requirements. The table should move up and down, or in and out of the bore, by operating the tumble switch. The table should not move beyond the boundaries shown in Fig. 4. The case-study is specified using a top-down design methodology. In other words, we first model the overall system at a higher level of abstraction in which we identify that the system consists of a controller and a user interface. Furthermore, a controller can run in the following three modes: *Init* mode in which the controller should place the table in the initial position; *Normal* mode in which the controller synchronises with the events of the tumble switch; *TTR* (Table Top Release) mode in which an operator is allowed to override the normal execution of the controller. Fig. 5 shows the model of the system at this level of abstraction. Throughout the complete description of this case-study, it is assumed that only the τ action is urgent. All other actions, which are the actions generated by the user interface, are non-urgent. This is modeled by $\nu_\tau(PSS)$ where *PSS* represents the automaton *PSS* shown in Fig 5.

User interface. The user interface consists of three input devices: the tumble switch, the table top release switch and the start button (See Fig. 6). The tumble switch has three positions: *MvUpOrIn*, *Neutral* and *MvDownOrOut*. The *MvUpOrIn* position is used to move the table either up or into the bore, the *MvDownOrOut* position is used to move the table down or out of the bore. When the switch is released, it returns to the neutral position, which enforces actuated (motorized) movement of the table to stop.

The TTR switch can be used to release the table top from the horizontal motor. When the switch is active, the horizontal movement of the table is uncontrolled by the system, so that an operator can manually move the table freely in the horizontal direction. Finally, the start button initializes the system.

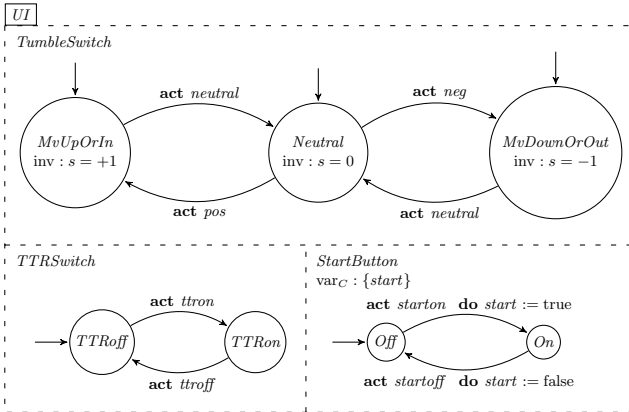


Fig. 6. User interface

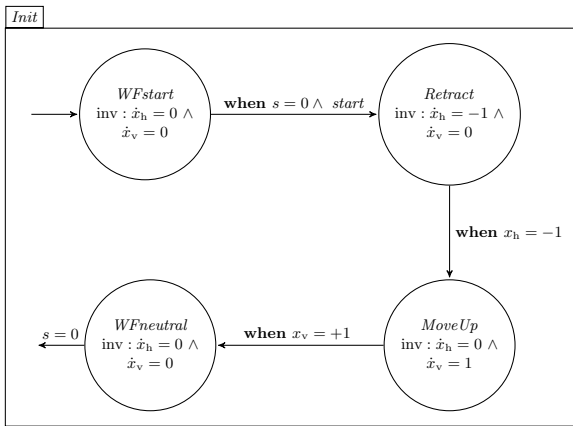


Fig. 7. Initialization

Initialization. In the *Init* mode, the position of the patient support system is initialized (Fig. 7). The position of the tumble switch needs to be neutral before initialization begins, and the movement is triggered by pressing the start button. The desired final position of the table is fully retracted and fully up. First, the table is retracted since this is always a safe movement. Then, when the table is fully retracted, the table is moved up until it reaches the top position. The initialization is complete when the tumble switch is in the neutral position, to prevent that the table starts moving immediately after initialization.

Normal mode. Initially, the system enters the normal mode with the table fully up and retracted, so in an up and out position (Fig. 8). In this intersection point between moving the table horizontally or vertically, holding the tumble switch in the *MvUpOrIn* position triggers horizontal movement of the table

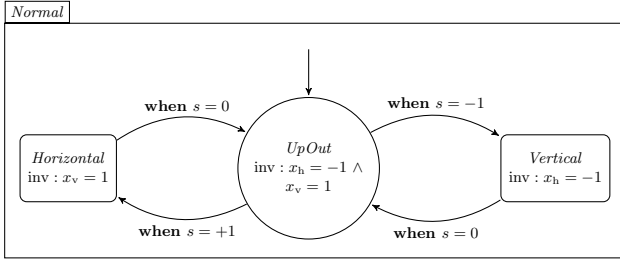


Fig. 8. Normal movement control

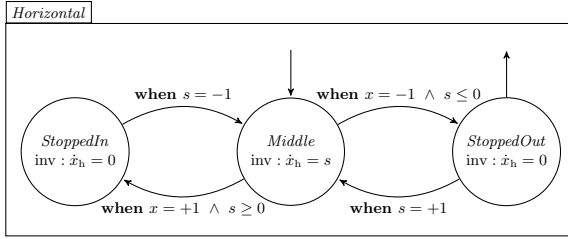


Fig. 9. Horizontal movement control

into the bore, whereas holding it in a *MvDownOrOut* position triggers vertical, downward movement.

A system requirement is that between switching from horizontal to vertical movement, and vice versa, the position of the tumble switch must be neutral. This to prevent the table from continuing movement unexpectedly in a different direction. Figure 9 show the horizontal movement of the system in more detail. The automaton for the vertical movement of the system is similar to the automaton drawn in Fig 9 and due to space reasons is not shown here.

6 Concluding Remarks

In this article we have presented the syntax and semantics of HCIF, which extends CIF with hierarchy in a compositional manner, so that only the SOS rules for an automaton and for the time transitions of parallel composition need to be adapted. As a result, we extended our previous work [6] to complete the hierarchical extension of CIF, which contains more involved concepts like invariants, synchronization, local variables and control variables.

We conjecture that we are able to transform a HCIF composition into a bisimilar CIF specification on the condition that the effective set of controlled variables and the effective dynamic type of the variables is independent of the active locations of the automata, and is thus statically defined. This condition is needed because in HCIF, the dynamic type of variables and the set of control variables can change per location, since the substructures of different locations may

have different dynamic types and different control variables. Future work includes proving that HCIF is more expressive than CIF, and defining the subset of HCIF that can be translated to CIF. These transformations are important to reuse existing tools for CIF, including model transformations.

In addition, we observe that a liberal interpretation of a HCIF automaton as an n -ary operator (where n represents the number of locations in the super-automaton) places our semantic rules within the congruence format of [15]. This means that the replacement of a sub-automaton by an equivalent one (modulo stateless bisimulation equivalence) will lead to an equivalent behavior of the super-automaton, which is a fundamental property for compositional reasoning.

Acknowledgements. The authors would like to thank Albert Hofkamp for helpful comments.

References

1. Alur, R., Dang, T., Esposito, J., Hur, Y., Ivančić, F., Kumar, V., Lee, I., Mishra, P., Pappas, G.J., Sokolsky, O.: Hierarchical modeling and analysis of embedded systems. *Proceedings of the IEEE* 91(1), 11–28 (2003)
2. Baeten, J., van Beek, D., Hendriks, D., Hofkamp, A., Agut, D.N., Rooda, J., Schiffelers, R.: Definition of the compositional interchange format. Technical Report Deliverable D1.1.2, Multiform (2010)
3. Basu, A., Bozga, M., Sifakis, J.: Modeling heterogeneous real-time components in bip. In: *Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods*, pp. 3–12. IEEE Computer Society, Washington, DC, USA (2006)
4. van Beek, D.A., Collins, P., Nadales, D.E., Rooda, J., Schiffelers, R.R.H.: New concepts in the abstract format of the compositional interchange format. In: Giua, A., Mahuela, C., Silva, M., Zaytoon, J. (eds.) *3rd IFAC Conference on Analysis and Design of Hybrid Systems*, Zaragoza, Spain, pp. 250–255 (2009)
5. van Beek, D.A., Reniers, M.A., Schiffelers, R.R.H., Rooda, J.E.: Foundations of a compositional interchange format for hybrid systems. In: Bemporad, A., Bicchi, A., Buttazzo, G. (eds.) *HSCC 2007*. LNCS, vol. 4416, pp. 587–600. Springer, Heidelberg (2007)
6. Beohar, H., Nadales Agut, D.E., van Beek, D.A., Cuijpers, P.J.L.: Hierarchical states in the compositional interchange format. *Electronic Proceedings in Theoretical Computer Science* 32, 42–56 (2010)
7. C4C consortium. Control for coordination of distributed systems (2008), <http://www.c4c-project.eu/>
8. Cuijpers, P.J.L., Reniers, M.A., Heemels, W.P.M.H.: Hybrid transition systems. Technical Report CS-Report 02-12, Eindhoven University of Technology, Department of Computer Science, The Netherlands (2002)
9. Frehse, G.: PHAVer: Algorithmic verification of hybrid systems past HyTech. In: Morari, M., Thiele, L. (eds.) *HSCC 2005*. LNCS, vol. 3414, pp. 258–273. Springer, Heidelberg (2005)
10. Harel, D.: Statecharts: A visual formalism for complex systems. *Science of Computer Programming* 8(3), 231–274 (1987)

11. Henzinger, T.A.: The theory of hybrid automata. In: Inan, M.K., Kurshan, R.P. (eds.) *Verification of Digital and Hybrid Systems*. NATO ASI Series F: Computer and Systems Science, vol. 170, pp. 265–292. Springer, New York (2000)
12. H. Highly-complex and networked control systems (2010), <http://www.hycon2.eu/>
13. HYCON Network of Excellence (2005), <http://www.ist-hycon.org/>
14. Lynch, N., Segala, R., Vaandrager, F.: Hybrid I/O automata revisited. In: Di Benedetto, M.D., Sangiovanni-Vincentelli, A.L. (eds.) *HSCC 2001*. LNCS, vol. 2034, pp. 403–417. Springer, Heidelberg (2001)
15. Mousavi, M.R., Reniers, M.A., Groote, J.F.: Notions of bisimulation and congruence formats for SOS with data. *Information and Computation* 200(1), 107–147 (2005)
16. MULTIFORM consortium. Integrated multi-formalism tool support for the design of networked embedded control systems MULTIFORM (2008), <http://www.multiform.bci.tu-dortmund.de>
17. Plotkin, G.D.: A structural approach to operational semantics. *Journal of Logic and Algebraic Programming* 60-61, 17–139 (2004)
18. Reynolds, J.C.: *Theories of programming languages*. Cambridge University Press, New York (1999)
19. The MathWorks, Inc., Simulink (2011), <http://www.mathworks.com>
20. Theunissen, R.J.M., Petreczky, M., Schiffelers, R.R.H., van Beek, D.A., Rooda, J.E.: Application of supervisory control synthesis to MRI scanners: improving evolvability. SE Report 2010-06, System Engineering Group, Department of Mechanical Engineering, Eindhoven university of technology, Eindhoven (2010)
21. Uselton, A.E., Smolka, S.A.: State Refinement in Process Algebra. Technical report, Stony Brook university, NY (1993)

Application of Model-Checking Technology to Controller Synthesis^{*}

Alexandre David¹, Jacob Deleuran Grunnet², Jan Jakob Jessen¹,
Kim Guldstrand Larsen¹, and Jacob Illum Rasmussen³

¹ Department of Computer Science, Aalborg University, Denmark
{adavid,kgl,jjjessen}@cs.aau.dk

² LAC Engineering, Hinnerup, Denmark
jag@lacengineering.com

³ Sanddru R&D, Nørresundby, Denmark
illum@sanddru.com

Abstract. In this paper we present two frameworks that have been implemented to link traditional model-checking techniques to the domain of control. The techniques are based on solving a timed game and using the resulting solution (a strategy) as a controller. The obtained discrete controller must fit within its continuous environment, which is modelled and taken care of in our frameworks. Our first technique does it by using *Matlab* to discretise the problem and then UPPAAL-TIGA to solve the obtained timed game. This is implemented as a toolbox. The second technique relies on the user defining a timed game model in UPPAAL-TIGA. Then the strategy is automatically imported in *Simulink* as an S-function for simulation and validation purposes. We demonstrate the effectiveness of these frameworks in different case-studies.

1 Introduction

The traditional control design cycle includes modelling, simulation, equation solving, and implementation. Modelling the environment and physical systems often means having to deal with non-linear or even hybrid models (mixing both discrete and continuous aspects) for which many of the standard control design methods are not easily applicable.

A major task for any control system designer is abstracting such models in to a form which is suitable for controller design e.g. by linearisation and when the controller design has been performed to simulations to validate the approximation and implementation of the control strategy. This is non-trivial and in this article we report on two prototypes for model-based design for optimal control using the controller synthesis tool UPPAAL-TIGA, *Matlab*, and its powerful toolbox *Simulink* [13].

The ultimate goal is to automate and unify the entire procedure such that the control system designer can perform modelling, synthesis and verification in a

^{*} Work supported by the MULTIFORM project FP7-ICT-2007-2.

single tool, while providing only the system specification and requirements. The first prototype is the toolbox called *PAHSCTRL* [6] that enables computation of piecewise-affine control laws for hybrid systems with non-deterministic discrete transitions. In particular, this can be used for fault-tolerant control [8]. The second prototype is a generalisation of the climate controller case-study [11] implemented in the form of Ruby scripts that are called from within *Matlab* to integrate seamlessly with *Simulink*.

In this paper we gather previous results obtained for both *PAHSCTRL* and UPPAAL-TIGA. In addition, we define a more general framework for linking UPPAAL-TIGA to *Simulink* and detail its implementation. We first give the background of timed games, then we present *PAHSCTRL*, and finally the framework for linking UPPAAL-TIGA to *Simulink*.

2 Controller Synthesis with Timed Game Automata

In our setting we use the model of timed game automata, an extension of timed automata, to define a game between two players: a controller and an environment. The goal is find a strategy for the controller player to meet a control objective for any move of the environment player. We refer to [2,5] for more details on the formalism, here we only summarise the important notions.

Let X be a finite set of real-valued variables called clocks. We note $\mathcal{C}(X)$ the set of constraints φ generated by the grammar: $\varphi ::= x \sim k \mid x - y \sim k \mid \varphi \wedge \varphi$ where $k \in \mathbb{Z}$, $x, y \in X$ and $\sim \in \{<, \leq, =, >, \geq\}$. $\mathcal{B}(X)$ is the subset of $\mathcal{C}(X)$ that uses only rectangular constraints of the form $x \sim k$.

Definition 1. A Timed Automaton (TA) [7] is a tuple $A = (L, l_0, \Sigma, X, E, Inv)$ where L is a finite set of locations, $l_0 \in L$ is the initial location, Σ is the set of actions, X is a finite set of real-valued clocks, $Inv : L \rightarrow \mathcal{B}(X)$ associates to each location its invariant and $E \subseteq L \times \mathcal{B}(X) \times \Sigma \times 2^X \times L$ is a finite set of transitions, where $t = (l, g, a, R, l') \in E$ represents a transition from the location l to l' , labelled by a , with the guard g , that resets the clocks in R . One special label τ is used to code the fact that a transition is not observable.

Definition 2. A Timed Game Automaton (TGA) [12] is a timed automaton G with its set of transitions E partitioned into controllable (E^c) and uncontrollable (E^u) actions. In addition, invariants are restricted to $Inv : L \rightarrow \mathcal{B}'(X)$ where \mathcal{B}' is the subset of \mathcal{B} using constraints of the form $x \leq k$.

Given a TGA G and a control property $\phi \equiv \mathcal{A} \phi_1 \mathcal{U} \phi_2$ (resp. $\mathcal{A} \phi_1 \mathcal{W} \phi_2$) of ATCTL, the *reachability* (resp. *safety*) control problem consists in finding a strategy f for the controller such that all the runs of G supervised by f satisfy the formula [4]. A strategy is a mapping from states to action to perform, an action being just to delay or to delay some time and to take a transition. The controller

¹ Here \mathcal{U} stands for the until operator and \mathcal{W} for the weak until operator. In the context of UPPAAL-TIGA, they have slightly different semantics than the usual operator in the sense that ϕ_1 should still be satisfied when reaching ϕ_2 .

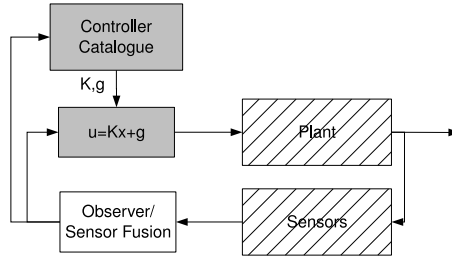


Fig. 1. The *PAHSCTRL* controller structure. Grey boxes are synthesised, the white box represents the part of the control system not handled and hatched boxes represent the physical plant and sensor systems.

synthesis problem is formulated in our setting as a timed game to solve and the resulting controller is the strategy obtained. We refer to strategies being winning when there is a strategy for the controller player to meet its control objective.

3 *PAHSCTRL*

3.1 Introduction

The focus of this toolbox and framework is to automatically generate controllers for piecewise-affine hybrid systems (PAHS). Using the proposed method, fault tolerant controllers are designed by modelling faults as uncontrollable events causing switches between discrete modes. The design method involves abstracting PAHS to discrete games and deriving controllers based on winning strategies for the game.

The idea of solving control problems by abstracting to discrete or timed games is not in itself new. The inspiration to use control to ensure that the system behaviour conforms to the discrete abstraction comes from [14], which demonstrates how controllers for discrete linear systems can be designed to conform to Linear Temporal Logics (LTL)-specifications.

Our method builds on advances in controller synthesis for affine systems on polytopes [10,9], where it is also suggested to design controllers for PAHS by abstraction. The toolbox presented here and detailed in [8] expands on these ideas, principally by adding (uncontrollable) external events, which can trigger transitions between modes.

The result is a *Matlab* toolbox capable of computing discrete game abstractions of PAHS and deriving control laws based on solutions to the discrete game. This enables computation of a type of *gain scheduling controller* where the control gains are adjusted based on the position in the state-space and the current fault condition. The resulting controller structure is shown in figure 1.

The toolbox implements and optimises the algorithms shown in [7] that details how a discrete game abstraction can be obtained for non-deterministic piecewise-affine hybrid system (PAHS). The game may be solved automatically

with respect to reachability properties using UPPAAL-TIGA. Assuming that a winning strategy exists for this game, it can be interpreted as a rule base that determines which control law to use in a given condition, and m-functions refine the result to affine control laws, thus synthesising the complete control system.

Our goal is to automate the procedure of finding control laws for hybrid systems with non-deterministic discrete state transitions. This is achieved by calculating a catalogue of affine control laws each acting on a subset of the state-space. Together they should guarantee a set of control requirements in the form of reach/avoid specifications.

In this framework, the user formalises the hybrid system and enters it inside *Matlab*. Then the *PAHSCTRL* tool generates the discrete game, whereas in the second framework the user has to make this model.

The toolbox is implemented in *Matlab* and consists of a number of m-functions designed to be easy to use but still exposing enough functionality that the toolbox can be used for different purposes. Compared to the algorithms presented in [7], a number of optimisations have been implemented along with new functionalities, in particular regarding refinement of control laws. The toolbox is detailed in [6] and can be found at <http://pahsctrl.polytekniker.dk>.

We demonstrate which kind of problems can be solved and we outline the solution strategy employed.

3.2 Problem Definition

Informally a PAHS as defined in *PAHSCTRL* is a discrete automaton where each location is associated with a continuous system. The locations are referred to as *modes* and the state space of each mode is partitioned into polytopes each associated with an affine system of the form

$$\dot{x} = Ax + Bu + C \quad (1)$$

where x is the state variable, u is the input and A, B, C are matrices of appropriate size.

The transitions between the modes in the discrete automata can be both controllable and uncontrollable meaning that they are taken by, respectively, the controller or the environment. Such transitions can be used to model faults or other externally triggered events that change the system dynamics.

The goal of the controller synthesis is therefore to compute a control strategy that ensures that a subset of the state space of one or more modes is reached while avoiding other subsets.

This is best illustrated with the example shown in figure 2. Not shown is the uncontrollable transition from mode 1 to 2 with an identity reset map. Mode 1 can be thought of as the nominal mode while mode 2 corresponds to a fault mode.

The “hole” in the partition is a subset of state space where no dynamics have been defined. This can be used as an alternative method for describing avoid sets, with the important difference that no dynamics are specified for the “hole” and no computational effort is spent on this region of state space.

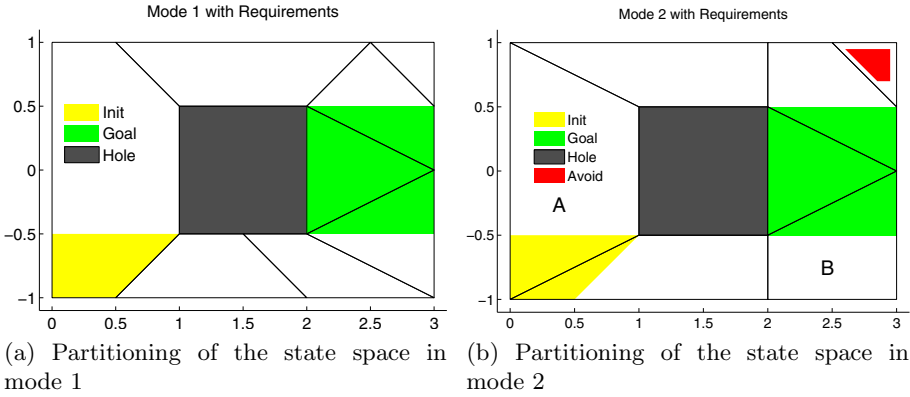


Fig. 2. Partitioning of the two modes including requirements specification given as polytope sets. There is an uncontrolled transition from mode 1 to mode 2, which can occur at any point in time.

The objective of the example is to reach the goal set while staying in the partitioned set and avoiding the avoid set.

The main algorithm consists of:

1. computing a discrete game abstraction of the hybrid system,
2. finding a solution to the game ensuring the control requirements, and
3. refining the solution to control laws of the form $u = Kx + g$.

This toolbox aims at solving 1) and 3) using *Matlab*, while leaving step 2) to established tools such as UPPAAL-TIGA.

3.3 Abstraction

During the abstraction, discrete equivalents of each polytope are computed. That is for each polytope defined on each mode one or more abstractions are computed to encode the possible actions of an affine controller as a discrete game.

The actions are encoded according to the ability of the controller to prevent the system from leaving the polytope through a given facet. The actions are labelled as follows.

Blockable. A control law exists that prevents exit through this facet

Uncontrollable. The facet is not blockable.

Controllable. The facet is blockable and a control law exists that can unblock the facet while ensuring that the polytope is left in finite time.

An example is presented in figure 3, showing a polytope with the controllable system directions indicated at the vertices. Dashed transitions are uncontrollable.

The discrete equivalents of a polytope are computed by solving for feasibility of linear matrix inequalities (LMI) based on the controllability to facet results presented in [9]. Each possible combination of facet labels corresponds to one LMI.

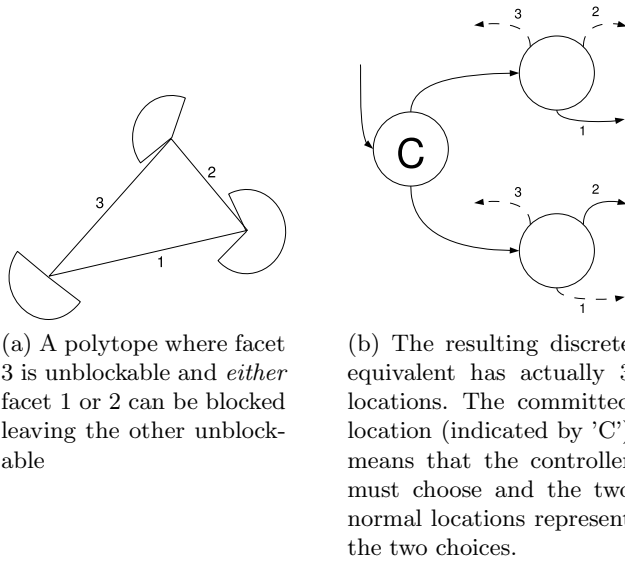


Fig. 3. A 2-dimensional polytope being converted to a discrete equivalent in a discrete game. In this case there are two possibilities which are merged via a committed location.

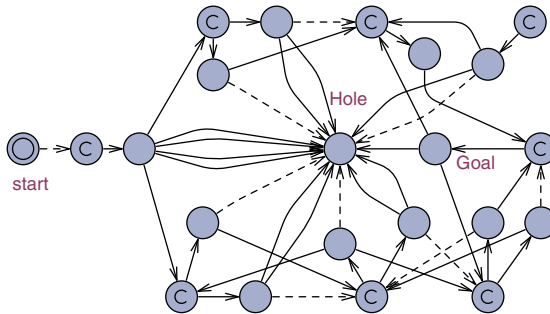


Fig. 4. A discrete game abstraction of mode 1. Each committed location is at the approximate spot of its corresponding polytope. The hole location denotes unpartitioned space.

3.4 Strategy

By computing discrete equivalents for all the polytopes of the PAHS and combining them, a discrete game abstraction can be obtained. In figure 4 the discrete abstraction for mode 1 of the example is shown.

With this simple example it is easy to find a winning strategy manually. The goal is to find a path from the start location to the goal location that avoids locations from where there exists a sequence of uncontrollable transitions to the hole location.

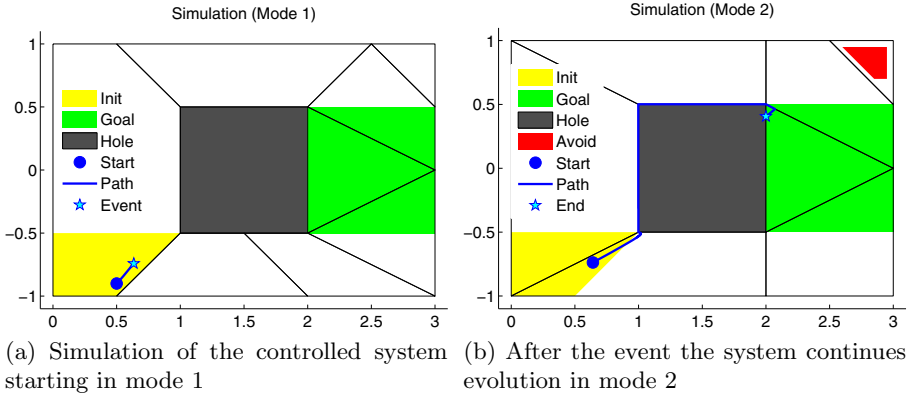


Fig. 5. Simulation of the example with the synthesised controller. The state ends at a fixed point just inside the goal set.

The toolbox uses UPPAAL-TIGA to find a winning strategy to the discrete abstraction, enabling a fully automated control synthesis.

3.5 Refinement

The affine control laws on the original PAHS are found by refining the winning strategy computed by UPPAAL-TIGA. The strategy determines which discrete equivalent is to be used and thus determines which LMI is used to limit the control law.

From the abstraction step it is known that the LMI chosen for each polytope is feasible and the refinement step is thus restricted to finding an optimal solution to each LMI. Combining these LMI solutions yields a controller catalogue, one controller for each polytope, which ensures that the state will reach the goal set in finite time. A simulation of the example using a controller generated by *PAHSCTRL* is shown in figure 5. The path goes around the hole on the border (left and then above).

4 Linking Uppaal-tiga to *Simulink*

4.1 Introduction

Our second framework provides an integrated and complete tool chain for modelling, synthesis, simulation, and automatic generation of executable code. The framework requires that two models of the control problem are provided: an abstract model in terms of a timed game and a complete, dynamic model in terms of a (non-linear) hybrid system.

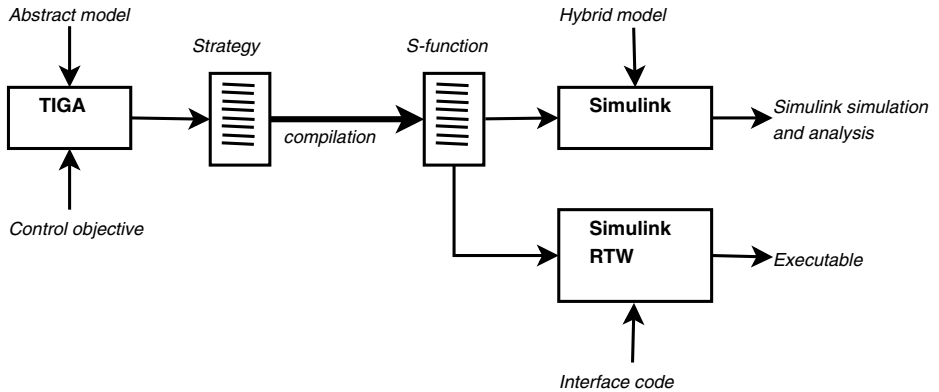


Fig. 6. Overview of the framework

Given the abstract (timed game) model together with logically formulated control and guiding objectives, UPPAAL-TIGA automatically synthesises a strategy which is directly compiled into an S-function².

Figure 6 shows an overview of the framework. It is based on our previous case-study of a climate controller for poultry and pig farms [11]. In this previous work, the humidity and heat transfer were described by their differential equations between zones in the farm. Then we simplified and discretised the model as a timed game automaton that was used to generate a controller automatically. That controller was then plugged into *Simulink* to validate through simulation using the non-linear model that the controller was able to control the climate as expected. It was possible to study its performance and, by choosing different control objectives, we could easily change the controller and simulate the new versions. The code generation was made possible through the *Simulink* real-time workbench. From the point where we have an S-function, we can simulate and generate real code. We have successfully redone this case-study [11] using our general framework instead of the custom translations. We recall that a controller was manually made taking into account only the temperature (not the humidity) and was found to be the same as the one generated by UPPAAL-TIGA. Then humidity was added to the model but this was too complex for the manual step. In addition, the objective function is given with weights on the temperature and humidity to optimise the criteria we want. This allows the generation of a series of controllers to simulate and validate their behaviours in *Simulink*. The goals of our extensions here are to i) integrate UPPAAL-TIGA and *Simulink*, and ii) to generalise the framework.

² S-function is a term used in *Simulink* for executable content that can be embedded into its block components. S-functions support multiple languages such as C and *Matlab* representation of the controller.

4.2 Work-Flow

In this framework the user formalises the environment and the physics of the system using classical differential equations. This is then abstracted in terms of timed game automata. As in [4] the continuous domain is discretised into intervals that correspond to clock constraints to model the dynamics. The abstract (discretised) model is entered in UPPAAL-TIGA. The model gives the possible moves for the environment and the controller players. The tool solves the game and generates a strategy (if possible) to meet a given control objective.

In parallel, the continuous model is entered in *Simulink* with a place-holder S-function that will act as the controller. Inputs and outputs for this block correspond to the UPPAAL-TIGA model. Using our translator we plug the generated discrete controller into *Simulink* to simulate it in its continuous environment. We note that it is now easy to change parameters in the model, generate new controllers and study their performances. In addition, using the *Simulink* real-time workbench allows us to generate real code for a given target platform.

4.3 Tool Integration

Figure 7 shows a more detailed view of the tool integration that we have implemented. The implementation is separated into one (internal) *Matlab* function that acts as the coordinator component and a Ruby script that makes the translation from a strategy to an S-function. The user defines a timed game automaton model in UPPAAL-TIGA together with a *Simulink* model that contains a block in which the user wishes to insert the generated controller from UPPAAL-TIGA. It is up to the user to define the input and the output variables. These inputs and outputs are defined in *Simulink* and their names must match the corresponding variables in the UPPAAL-TIGA model. The user should make sure that the desired property is satisfied to obtain a strategy. Then the user calls the *Matlab* function that

1. calls UPPAAL-TIGA to generate the strategy,
2. extracts the inputs and outputs from the *Simulink* model and generates input and output files,
3. calls the Ruby script that translates the strategy together with the declaration files of inputs and outputs into an S-function,
4. and calls the *Matlab* C-compiler to compile the generated S-function and imports the binary into *Simulink*.

The *Simulink* model can now be simulated with the generated controller or it can be used with the real-time workbench to generate code from the S-function.

4.4 Mapping to *Simulink*

To have the generated strategy (from UPPAAL-TIGA) work in *Simulink*, the models need to obey a few constraints. First *Simulink* will play the uncontrollable

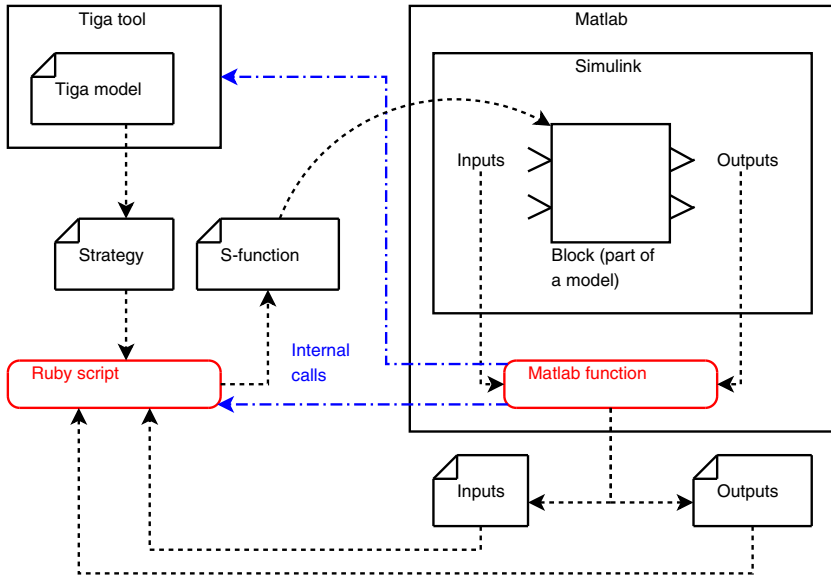


Fig. 7. Integration of UPPAAL-TIGA and *Simulink*

transitions but they should not change location in our model, only integer variables. This models the input from the environment. In our example we define that the temperature variables (in fact the indices) are the inputs. They are allowed to change according to our model in UPPAAL-TIGA and the model in *Simulink* should match this behaviour.

Second, we need to define outputs from our controller to *Simulink*. The controller can change its own locations, variables, and clock. We define that some of these variables are used as output to *Simulink*.

Finally, time is discretised by *Simulink* with some resolution. Our strategy is ultimately transformed to an S-function that is in fact a decision function with some added code to make the interface between *Simulink* variables and state variables of the controller. Clocks are incremented at every call of the function and the strategy decides what to do at every *tick* (possibly just wait).

The transformation from our strategy (mapping from states to action) is done as if-statements that transform the updates in the timed game automata into statements. Furthermore, if we had used functions in the model, they are evaluated and transformed into simple assignment statements, which results in a strategy devoid of functions in UPPAAL-TIGA syntax. This is possible because such functions have their output solely determined by the discrete state they are evaluated on and states are known in the strategies. The generated code starts by accessing the input and output ports of *Simulink*. Incrementing the clocks is then done after taking the actions to prepare for the next call of the function. The trade-off in this solution is that we let the user test clock values for zero upon the first call but afterwards we will never get zeros again since the

discretisation forces a minimal time between the action and the next time we can read inputs and take a decision again. The user will be able to simulate the generated strategy and see if the system is stable with the chosen parameters in spite of the discretisation.

4.5 Methodology and Example

The first task is to abstract the physical model to a timed game automaton. In our extension, *timed* controllers are supported³ and they are integrated in *Simulink* by discretising time. The abstraction here consists in mapping the continuous behaviour of a system to the time dimension and to make control decision based on chosen intervals. The goal is to keep the abstraction as coarse as possible to simplify the controller but in principle we could discretise with a fine granularity and model the behaviour as precisely as we want.

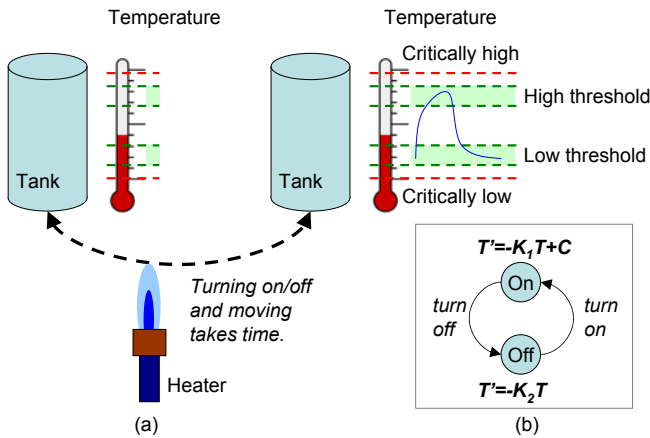


Fig. 8. The 2-tank example. One heater can heat one tank at a time and moving the heater between the tanks takes time (a). The temperature of the tanks should stay within an acceptable range. The temperature is modelled by the simple hybrid system in (b) with two states associated with differential equations.

To illustrate the modelling step, we consider a 2-tank example as shown in figure 8(a). The idea is to maintain the temperature of two tanks containing some liquid within some specified bounds. We have one heater that can be used to heat either one of the two tanks, but changing tank takes time. The temperature of the tanks should be kept between a safe middle range and in our abstraction we consider critical low or high temperatures that we do not want to reach and two ranges of temperatures that are observable by our controller. These serve as low and high thresholds as shown in the figure. The hybrid model of the dynamics

³ This is in contrast to our previous work where only untimed strategies were supported by our framework.

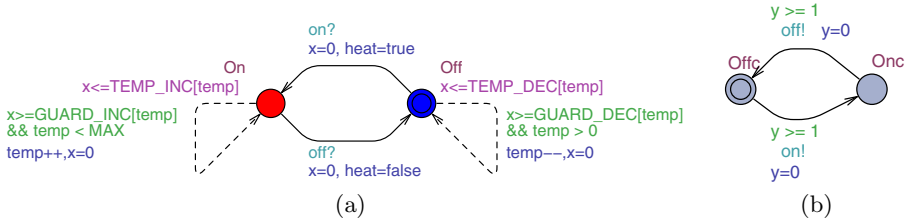


Fig. 9. The model of the 2-tank example in UPPAAL-TIGA

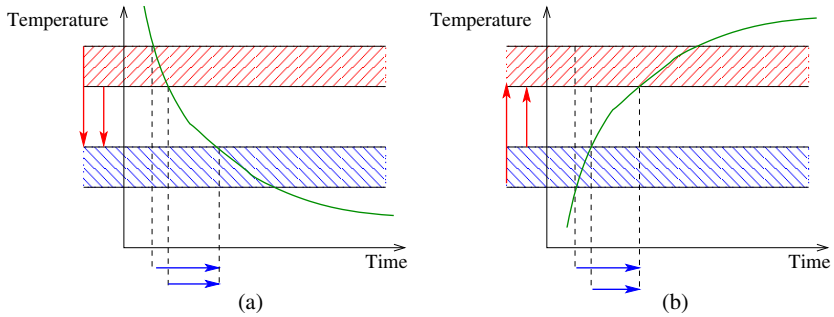


Fig. 10. Principle for mapping temperature changes to time when the temperature is decreasing (a) or increasing (b). We obtain a lower and an upper bound on time for changing temperature range.

is simple here as shown in figure 8(b). We have a state machine (for each tank) with two states to denote when the heater is on or off with associated differential equations to describe how the temperature changes. T is the temperature, K_1 , K_2 , and C are constants.

We model this system in UPPAAL-TIGA with one process per tank and one for the controller. Figure 9 shows the templates for the tank and the controller. The tank automaton (Fig. 9(a)) reflects the two states of the heater being on and off and a clock x is used to measure time.

Temperature changes are then mapped to time intervals and the model is designed to take uncertainties into account. Figure 10 shows the principle. In Fig. 10(a) the temperature decreases from somewhere from the high observable range to the lower one. We derive a lower and upper bound on time for detecting the state change. Similarly we derive time bounds when the temperature increases in Fig. 10(b). The lower bounds are modelled by the guards ($x \geq \text{GUARD_DEC}[\text{temp}]$ and $x \geq \text{GUARD_INC}[\text{temp}]$ when the temperature is decreasing or increasing) and the upper bounds are the invariants ($x \leq \text{TEMP_DEC}[\text{temp}]$ and $x \leq \text{TEMP_INC}[\text{temp}]$ depending on heating). The model is designed to discretise an arbitrary number of such observable ranges and we make experiments with two and three such ranges. The controller (Fig. 9(b)) models that it can turn a heater on or off with a constraint on time.

Given some dynamic model in *Simulink*, we extract the time ranges that we insert in UPPAAL-TIGA. We first make the experiments with the following ranges:

- Above 100, temperature is critical high (`temp=3`).
- Between 70 and 90, temperature is high (`temp=HIGH=2`).
- Between 40 and 60, temperature is low (`temp=LOW=1`).
- Below 30, temperature is critical low (`temp=0`).

The corresponding time intervals in the models are declared as follows⁴:

```
const int TEMP_INC[temperature_t] = { 0, 6, 7, 0 };
const int TEMP_DEC[temperature_t] = { 0, 18, 10, 0 };
const int GUARD_INC[temperature_t] = { 0, 2, 2, 0 };
const int GUARD_DEC[temperature_t] = { 0, 7, 3, 0 };
```

The system is initialised with tank 1 at 55 degrees and tank 2 at 75 degrees, which corresponds to `temp` being 1 and 2. We note that the model detects changes of temperature so the actual range of temperature depends on the state (heating or not). We ask for the following control objectives:

```
control: A[] temp1>=LOW && temp1<=HIGH && temp2>=LOW && temp2<=HIGH
control: A[] temp1>=LOW && temp1<=LOW && temp2>=LOW && temp2<=HIGH
control: A[] temp1>=LOW && temp1<=HIGH && temp2>=HIGH && temp2<=HIGH
```

The two first objectives are met and UPPAAL-TIGA generates strategies that we insert in *Simulink*. The third one is not due to the constraints of the model (there is no winning strategy for this game). We plot in figure 11 the result of the simulations for the first (a) and second (b) properties. We note that first, having `temp1` staying at `LOW` depends much on the timing parameters because there is no other observable range that the controller can use. Stability of the simulated system depends on the uncertainties used in the model. Second, UPPAAL-TIGA generates one arbitrary strategy that is only guaranteed to meet a control objective in the model. The simulation allows the user to evaluate its performance. With the loose specification of the first property, the controller chooses to keep one tank at a high temperature and the second one at a low temperature. The choice is natural w.r.t. their initial conditions. For the second property the controller chooses to keep both tanks in the same range even though the previous strategy could have been enough. This is not a bug in the controller since the temperatures both follow their specifications.

We repeat the experiments by defining the following ranges instead:

- Above 100, temperature is critical high (`temp=4`).
- Between 80 and 90, temperature is high (`temp=HIGH=3`).
- Between 60 and 70, temperature is good (`temp=GOOD=2`).
- Between 40 and 50, temperature is low (`temp=LOW=1`).
- Below 30, temperature is critical low (`temp=0`).

⁴ The 0 entries do not matter since we want to avoid these states. The parameters in *Simulink* are arbitrary, the important point is to derive our constants from them.

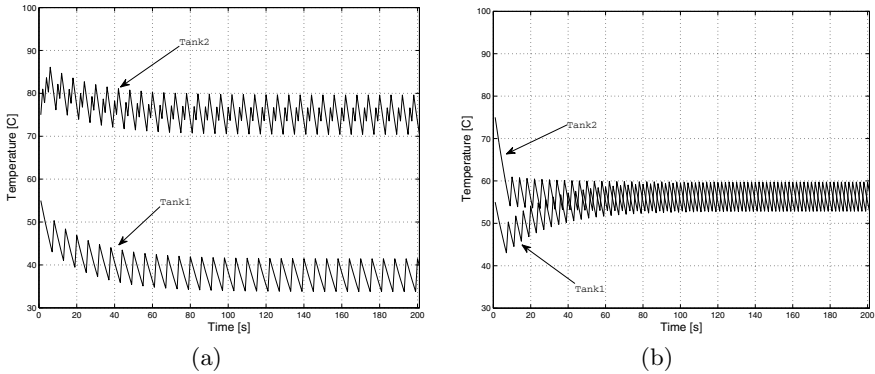


Fig. 11. Simulation results with two observable ranges, one simulation for each control objective

The corresponding declaration of parameters is:

```
const int TEMP_INC[temperature_t] = { 0, 4, 5, 5, 0 };
const int TEMP_DEC[temperature_t] = { 0, 13, 9, 7, 0 };
const int GUARD_INC[temperature_t] = { 0, 2, 2, 2, 0 };
const int GUARD_DEC[temperature_t] = { 0, 7, 4, 3, 0 };
```

We update the initial temperatures to be 65 and 85 for the two tanks with the corresponding temp being 2 and 3. We check for the following control objectives:

```
control: A[] temp1>=LOW && temp1<=HIGH && temp2>=LOW && temp2<=HIGH
control: A[] temp1>=LOW && temp1<=GOOD && temp2>=GOOD && temp2<=HIGH
control: A[] temp1>=LOW && temp1<=GOOD && temp2>=HIGH && temp2<=HIGH
```

Similarly the two first properties are satisfied but not the third one. We show the result of the simulation in figure 12. For the first property the controller chooses to keep both tanks within the same (large) range of temperatures. The second property results in separating the temperatures, as was the intention. We also experienced strategies in our experiments that would be similar to Fig. 11 (b) and still meet their control objectives.

The attentive reader would notice that for Fig. 11 (a) and Fig. 12 (b) the actual simulated temperature gets below 40 degrees though still above 30 degrees. The difference in the interpretation of the control objective comes from the fact that there is no temperature in the game model and the resulting controller uses the threshold “bands” as observations in a manner similar to [3] by detecting entering and leaving observations. The discretized controller takes decisions when crossing 40 degrees and never observes the temperature falling below 30 degrees. The parameters of these models would need to be refined to take decisions when crossing 50 degrees instead, which can be achieved by asking a different control objective.

We showed a methodology in this case-study to go from a hybrid model to a time model to generate a discrete controller. The approach matches the

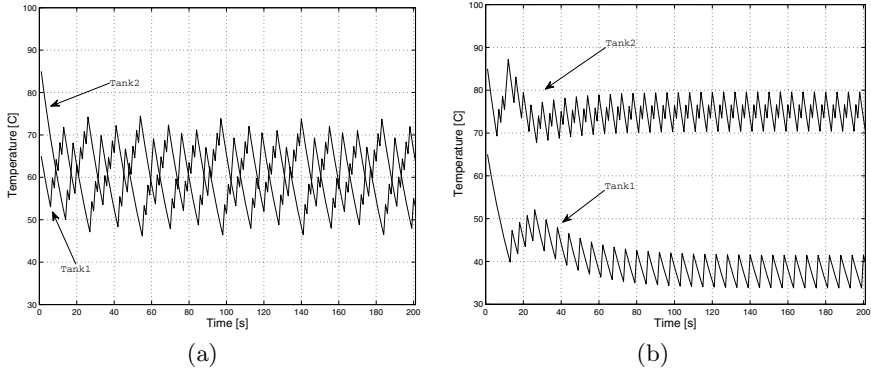


Fig. 12. Simulation results with three observable ranges, one simulation for each control objective

reality of having sensors that will detect changes (here of temperature) with some precision. The approach shows promising results.

5 Conclusion and Future Works

We have presented two frameworks that can be used to generate hybrid controllers and bridge the gap between control theory and its implementation on real hardware. Our case-studies show the viability of these approaches. Common for both methods is the use of (timed) game abstractions in order to get the problems on a computational tractable form. The *PAHSCTRL* toolbox enables automatic abstraction and refinement to and from discrete game form while the *UPPAAL-TIGA-Simulink* framework can simulate, solve and generate code for timed games.

Future works include how to merge the first approach with the second one to get the complete work-flow within *Simulink* for simulation and code generation purposes. The first framework generates models and is using only *Matlab* while the second framework takes advantage of *Simulink* but requires a manually constructed model. These approaches are complementary. In addition, *UPPAAL-TIGA* can represent strategies as multi-terminal decision diagrams and output them as pseudo-code in a different format. This could be used to generate more compact and efficient code.

References

1. Alur, R., Dill, D.: A theory of timed automata. *Theoretical Computer Science* 126(2), 183–235 (1994)
2. 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)

3. 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)
4. 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)
5. Chatain, T., David, A., Larsen, K.G.: Playing games with timed games. In: Giua, A., Mahulea, C., Silva, M., Zaytoon, J. (eds.) Preprints of the 3rd IFAC Conference on Analysis and Design of Hybrid Systems, pp. 238–243 (2009)
6. Grunnet, J.D., Bak, T., Bendtsen, J.D., Ankersen, F.: PAHSCTRL - a control synthesis toolbox for piecewise-affine hybrid systems. In: Proceedings of the 2009 European Control Conference. IEEE, Los Alamitos (2009)
7. Grunnet, J.D., Bak, T., Bendtsen, J.D., Larsen, J.A.: Discrete game abstraction for fault tolerant control synthesis. In: Proceedings of IEEE CACSD 2008 (2008)
8. Grunnet, J.D., Bendtsen, J.D., Bak, T.: Automated fault tolerant control synthesis based on discrete games. In: Proceedings of the 48th IEEE Conference on Decision and Control. IEEE, Los Alamitos (2009)
9. Habets, L., van Schuppen, J.H.: Control to facet problems for affine systems on simplices and polytopes - with applications to control of hybrid systems. In: Proc. 44th IEEE CDC (2005)
10. Habets, L.C.G.J.M., Collins, P.J., van Schuppen, J.H.: Reachability and control synthesis for piecewise-affine hybrid systems on simplices. IEEE Transactions on Automatic Control 51, 938–948 (2006)
11. 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)
12. Maler, O., Pnueli, A., Sifakis, J.: On the synthesis of discrete controllers for timed systems. In: Mayr, E.W., Puech, C. (eds.) STACS 1995. LNCS, vol. 900, pp. 229–242. Springer, Heidelberg (1995)
13. Mathworks. Simulink (2010)
14. Tabuada, P., Pappas, G.J.: Linear time logic control of discrete-time linear systems. IEEE Transactions on Automatic Control 51, 1862–1877 (2006)

Testing Real-Time Systems under Uncertainty

Alexandre David, Kim Guldstrand Larsen, Shuhao Li,
Marius Mikucionis, and Brian Nielsen

Center for Embedded Software Systems (CISS)
Aalborg University

Selma Lagerlöfs Vej 300, DK-9220 Aalborg, Denmark
{adavid, kgl, li, bnielsen, marius}@cs.aau.dk

Abstract. Model-based testing is a promising technique for improving the quality of testing by automatically generating an efficient set of provably valid test cases from a system model. Testing embedded real-time systems is challenging because it must deal with timing, concurrency, processing and computation of complex mixed discrete and continuous signals, and limited observation and control. Whilst several techniques and tools have been proposed, few deals systematically with models capturing the indeterminacy resulting from concurrency, timing and limited observability and controllability. This paper proposes a number of model-based test generation principles and techniques that aim at efficient testing of timed systems under uncertainty.

1 Introduction

Testing embedded real-time systems is challenging because it must deal with timing, processing and computation of complex mixed discrete and continuous signals, limited observation and control, and concurrency.

First, testing must evaluate the timeliness of the implementation under test (IUT), i.e., the tester must execute input stimuli with different timings, and evaluate response times against the specified tolerances. Second, the system performs complex computations on continuous signals that are observed through (potentially imprecise) sensors, sampled and discretized, implying that the resulting internal state and output values are not completely fixed, but should be accurate within some required bound. Third, because the system is embedded it is often problematic to create a test harness that gives a tester full observability and controllability of the internal state of the system. Sometimes testing has to be carried out via imprecise sensors and actuators. Finally, embedded systems are inherently highly concurrent and indeterminate because they need to control multiple simultaneous activities and detect and react to a multitude of events. This implies that executing the same input sequence may result in different output sequences, and that test cases in general need to branch out and be adaptive to uncertainty in the system's responses.

Formal specification models therefore tend to use non-determinism in many ways:

- models must be able to capture the concurrent behavior of real system implementations.
- models of large and complex systems are themselves constructed as parallel composition of models of its components.

- non-determinism is used in models to abstract away implementation details or aspects that are unknown or are too complex to fully model, e.g. details of data-representation and computation algorithms, or the state of pipelines, caches, bus-arbitration, and internal scheduling and resource allocation decisions.
- models should not capture one concrete implementation in all details, but specify its requirements, i.e. the permissible set of implementations. In particular with respect to timing, a model should capture requirements like the system should respond *within* x milli-seconds, not that it responds after precisely x milli-seconds.
- the system makes imperfect observations of its environment because of limitations on the precision with which it measures physical quantities (especially time), and the states/actions that are observable/controllable to the system in the first place. Dually, it makes imperfect output due to imprecise actuators. These limitations also apply to a tester (itself being a computer component) of the system.

Model-based testing is a promising technique for improving the quality of testing by automatically generating an efficient set of provably valid test cases from a system model. Research in the last decade has resulted in several approaches and mature tools. These are attracting interest by industry, and are being adopted by advanced developers. Whilst several techniques and tools have been proposed, few deal systematically with timing and limited observability and controllability. This paper proposes a number of model-based test generation techniques that aim at enabling efficient testing of timed systems under uncertainty. Given that efficient underlying state-exploration algorithms are very complex, another goal is to show how these techniques may be realized by existing tools.

2 Preliminaries

The ingredients for (formal) model-based testing is a (black-box) IUT, a specification model, a conformance relation defining which implementation behaviors are correct relative to the specification, and a test criterion defining which test cases should be generated.

We will assume that specifications and implementations can be understood as timed input/output transition systems (TIOTS), and use timed automata and timed game automata (with TIOTS semantics) to model the systems. Specifically, we use UPPAAL syntax and semantics. As conformance relation we use the relativized timed input/output conformance relation. A relativized conformance relation makes it possible to specify an explicit environment model for the IUT that defines the behavior of the context of the IUT. A test generator will use this to restrict the behavior considered for test cases: For embedded systems it is particularly important that the generated test cases are actually feasible in the technical and physical context of the system. As test criterion we use explicit test purposes. These ingredients are formalized in the following sections.

2.1 Timed I/O Transition Systems

A timed input/output transition system (TIOTS) is a transition system where actions have been classified as inputs or outputs, and where dedicated delay labels model the

progress of time. In our case we use the set of positive real-numbers to model time. Below we also extend commonly used notation for labeled transition systems to TIOTS.

Definition of TIOTS. We assume a given set of actions A partitioned into two disjoint sets of output actions A_{out} and input actions A_{in} . In addition we assume that there is a distinguished unobservable action $\tau \notin A$. We denote by A_τ the set $A \cup \{\tau\}$.

A *timed I/O transition system* (TIOTS) \mathcal{S} is a tuple $(S, s_0, A_{in}, A_{out}, \rightarrow)$, where

- S is a set of states, $s_0 \in S$,
- and $\rightarrow \subseteq S \times (A_\tau \cup \mathbb{R}_{\geq 0}) \times S$ is a transition relation satisfying the usual constraints of *time determinism* (if $s \xrightarrow{d} s'$ and $s \xrightarrow{d} s''$ then $s' = s''$), *time additivity* (if $s \xrightarrow{d_1} s'$ and $s' \xrightarrow{d_2} s''$ then $s \xrightarrow{d_1+d_2} s''$), and *zero-delay* (for all states $s \xrightarrow{0} s$). $d, d_1, d_2 \in \mathbb{R}_{\geq 0}$, and $\mathbb{R}_{\geq 0}$ denotes non-negative real numbers.

Let $a, a_{1..n} \in A$, $\alpha \in A_\tau \cup \mathbb{R}_{\geq 0}$, and $d, d_{1..n} \in \mathbb{R}_{\geq 0}$. We write $s \xrightarrow{\alpha}$ iff $s \xrightarrow{\alpha} s'$ for some s' . A run ρ of \mathcal{S} starting in state s_1 is a finite (or infinite) path in the timed transition system, $s_1 \xrightarrow{\alpha_1} s_2 \xrightarrow{\alpha_2} s_3 \xrightarrow{\alpha_3} \dots s_n$. Let $\text{Visits}(\rho)$ denote the set of states $s_1 \dots s_n$ traversed by ρ , and let $\text{Runs}(s)$ denote all runs starting in s . A run is *maximal* if $s_n \not\xrightarrow{\alpha}$. $\text{MaxRuns}(s)$ denotes the set of all maximal runs from state s . We use \Longrightarrow to denote the τ -abstracted transition relation such that $s \xrightarrow{a} s'$ iff $s \xrightarrow{\tau^*} \xrightarrow{a} \tau^* s'$, and $s \xrightarrow{d} s'$ iff $s \xrightarrow{\tau^*} \xrightarrow{d_1} \tau^* \xrightarrow{d_2} \tau^* \dots \xrightarrow{\tau^*} \xrightarrow{d_n} \tau^* s'$ where $d = d_1 + d_2 + \dots d_n$. We extend \Longrightarrow to sequences in the usual manner.

\mathcal{S} is *strongly input enabled* iff $s \xrightarrow{i}$ for all states s and for all input actions i . It is *weakly input enabled* iff $s \xRightarrow{i}$ for all states s and for all input actions i . We assume that input actions (seen from the system point of view) are controlled by the environment and outputs are controlled by the system. An input enabled system cannot refuse input actions. However it may decide to ignore the input by executing a transition that results in the same state.

\mathcal{S} is *non-blocking* iff for any state s and any $t \in \mathbb{R}_{\geq 0}$ there is a timed output trace $\sigma = d_1 o_1 \dots o_n d_{n+1}$, $o_i \in A_{out}$, such that $s \xrightarrow{\sigma}$ and $\sum_i d_i \geq t$. Thus \mathcal{S} will not block time in any input enabled environment. This property ensures that a system will not force or rush its environment to deliver an input, and vice versa, a non-blocking environment will never force outputs from the system. Time is common for both the system and its environment, and neither controls it.

To model potential implementations it is useful to define the properties of *isolated outputs* and *determinism*. \mathcal{S} is deterministic if for all delays or actions $\alpha \in A_\tau \cup \mathbb{R}_{\geq 0}$, and all states s , whenever $s \xrightarrow{\alpha} s'$ and $s \xrightarrow{\alpha} s''$ then $s' = s''$. That is, the successor state of an action is always uniquely known. We say that \mathcal{S} has *isolated outputs* if whenever $s \xrightarrow{o}$ for some output action o , then $s \not\xrightarrow{\tau}$ and $s \not\xrightarrow{d}$ for all $d > 0$ and whenever $s \xrightarrow{o'}$ then $o' = o$. A system with isolated outputs will only offer one output at a time, and will never retract an offered output by performing internal actions or delays.

Finally, a TIOTS exhibits *output urgency* iff whenever an output (or τ) is enabled, it will occur immediately, i.e., whenever $s \xrightarrow{\alpha}$, $\alpha \in A_{out} \cup \{\tau\}$ then $s \xrightarrow{d}$, $d \in \mathbb{R}_{\geq 0}$. An output urgent system will deliver the output immediately when ready.

An observable *timed trace* $\sigma \in (A \cup \mathbb{R}_{\geq 0})^*$ is of the form $\sigma = d_1 a_1 d_2 \dots a_k d_{k+1}$. We define the observable timed traces $\text{TTr}(s)$ of a state s as:

$$\text{TTr}(s) = \{\sigma \in (A \cup \mathbb{R}_{\geq 0})^* \mid s \xrightarrow{\sigma}\}$$

For a state s (and subset $S' \subseteq S$) and a timed trace σ , s After σ is the set of states that can be reached after σ :

$$s \text{ After } \sigma = \{s' \mid s \xrightarrow{\sigma} s'\}, \quad S' \text{ After } \sigma = \bigcup_{s \in S'} s \text{ After } \sigma$$

The set $\text{Out}(s)$ of observable outputs or delays from states $s \in S' \subseteq S$ is defined as:

$$\text{Out}(s) = \{a \in A_{out} \cup \mathbb{R}_{\geq 0} \mid s \xrightarrow{a}\}, \quad \text{Out}(S') = \bigcup_{s \in S'} \text{Out}(s)$$

TIOTS Composition. Let $\mathcal{S} = (S, s_0, A_{in}, A_{out}, \rightarrow)$ and $\mathcal{E} = (E, e_0, A_{out}, A_{in}, \rightarrow)$ be TIOTSSs. Here E is the set of environment states and the set of input (output) actions of \mathcal{E} is identical to the output (input) actions of \mathcal{S} . The parallel composition of \mathcal{S} and \mathcal{E} forms a *closed system* $\mathcal{S} \parallel \mathcal{E}$ whose observable behavior is defined by the TIOTS $(S \times E, (s_0, e_0), A_{in}, A_{out}, \rightarrow)$ where \rightarrow is defined as

$$\frac{s \xrightarrow{a} s' \quad e \xrightarrow{a} e'}{(s, e) \xrightarrow{a} (s', e')} \quad \frac{s \xrightarrow{\tau} s'}{(s, e) \xrightarrow{\tau} (s', e)} \quad \frac{e \xrightarrow{\tau} e'}{(s, e) \xrightarrow{\tau} (s, e')} \quad \frac{s \xrightarrow{d} s' \quad e \xrightarrow{d} e'}{(s, e) \xrightarrow{d} (s', e')}$$

2.2 Timed Automata

Timed automata [2] is an expressive and popular formalism for modeling real-time systems. Essentially a timed automaton is an extended finite state machine equipped with a set of real-valued clock-variables that track the progress of time and that can guard when transitions are allowed.

Definition of Timed Automata. Let X be a set of $\mathbb{R}_{\geq 0}$ -valued variables called *clocks*. Let $\mathcal{G}(X)$ denote the set of *guards* on clocks being conjunctions of constraints of the form $x \bowtie c$, and let $\mathcal{U}(X)$ denote the set of *updates* of clocks corresponding to sets of statements of the form $x := c$, where $x \in X$, $c \in \mathbb{N}$, and $\bowtie \in \{\leq, <, =, >, \geq\}$. A *timed automaton* (TA) over actions A and clocks X is a tuple (L, ℓ_0, A, X, I, E) , where

- L is a set of locations, $\ell_0 \in L$ is an initial location,
- $I : L \rightarrow \mathcal{G}(X)$ assigns invariants to locations, and
- E is a set of edges such that $E \subseteq L \times \mathcal{G}(X) \times A_\tau \times \mathcal{U}(X) \times L$.

We write $\ell \xrightarrow{g, \alpha, u} \ell'$ iff $(\ell, g, \alpha, u, \ell') \in E$. The semantics of a TA is defined in terms of a TIOTS over states of the form $s = \langle \ell, \bar{v} \rangle$, where ℓ is a location and $\bar{v} \in \mathbb{R}_{\geq 0}^X$ is a clock valuation satisfying the invariant of ℓ . Intuitively, a timed automaton can either progress by executing an edge or by remaining in a location and letting time pass:

$$\frac{\forall d' \leq d. I_\ell(d')}{\langle \ell, \bar{v} \rangle \xrightarrow{d} \langle \ell, \bar{v} + d \rangle} \quad \frac{\ell \xrightarrow{g, \alpha, u} \ell' \wedge g(\bar{v}) \wedge I_{\ell'}(\bar{v}'), \bar{v}' = u(\bar{v})}{\langle \ell, \bar{v} \rangle \xrightarrow{\alpha} \langle \ell', \bar{v}' \rangle}$$

In delaying transitions, $\langle \ell, \bar{v} \rangle \xrightarrow{d} \langle \ell, \bar{v} + d \rangle$, the values of all clocks of the automaton are incremented by the amount of the delay d , denoted $\bar{v} + d$. The automaton may delay in a location ℓ as long as the invariant I_ℓ for that location remains true. Discrete transitions $\langle \ell, \bar{v} \rangle \xrightarrow{\alpha} \langle \ell', \bar{v}' \rangle$ correspond to execution of edges $(\ell, g, \alpha, u, \ell')$ for which the guard g is satisfied by \bar{v} , and for which the invariant of the target location $I_{\ell'}$ is satisfied by the updated clock valuation \bar{v}' . The target state's clock valuation \bar{v}' is obtained by applying clock updates u on \bar{v} . A TA is *M-bounded*, if no clock value exceeds $M \in \mathbb{N}$.

UPPAAL Timed Automata. Throughout the paper we use UPPAAL syntax to illustrate TA. It allows construction of large models by composing TA in parallel and lets these communicate using shared discrete and clock variables and synchronize (rendezvous-style) on complementary input and output actions, as well as broadcast actions.

Initial locations are marked using a double circle. Edges are by convention labeled by the triple: guard, action, and assignment in that order. The internal τ -action is indicated by an absent action-label. Committed locations are indicated by a location with an encircled "C". A committed location must be left immediately by the next transition taken in the system. Finally, bold-faced clock conditions placed under locations are location invariants. In addition to clocks, UPPAAL also allows integer variables to be used in guards and assignments, and supports a safe subset of C-like data types and code in assignments and guards.

Figure 1 shows a TA modeling the behavior of a simple light-controller. The user interacts with the controller by touching a touch sensitive pad. The light has three intensity levels: OFF, DIM, and BRIGHT. Depending on the timing between successive touches (recorded by the clock x), the controller toggles the light levels.

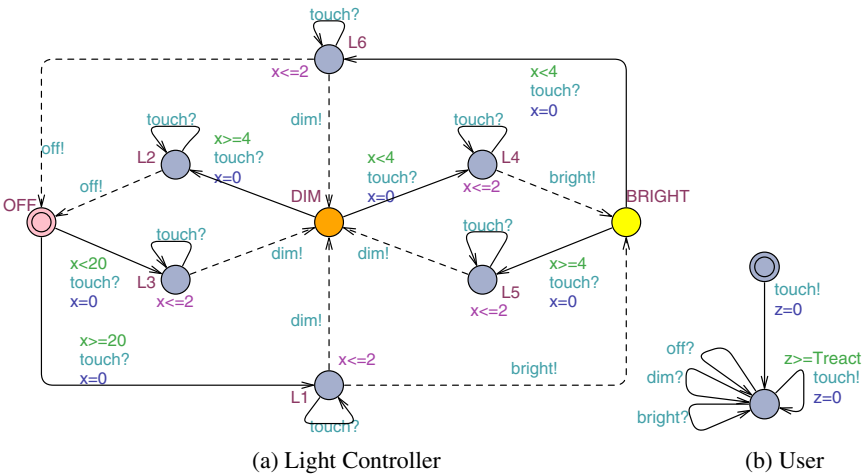


Fig. 1. Simple Light Controller Timed (Game) Model

For example, in dimmed state, if a second touch is made quickly (before the switching time 4 time units) after the touch that caused the controller to enter dimmed state (from either off or bright state), the controller increases the level to bright. Conversely, if the second touch happens after the switching time, the controller switches the light off. If the light controller has been in the off state for a long time (more than 20 time units), it should reactivate by going either to the dim level, or directly to bright level.

The light controller can perform the run $\langle \text{OFF}, x = 0 \rangle \xrightarrow{5} \langle \text{OFF}, x = 5 \rangle \xrightarrow{\text{touch}^?} \langle \text{L3}, x = 0 \rangle \xrightarrow{\text{dim}!} \langle \text{DIM}, x = 0 \rangle \xrightarrow{3.14} \langle \text{DIM}, x = 3.14 \rangle \xrightarrow{\text{touch}^?} \langle \text{L4}, x = 0 \rangle \xrightarrow{0.5} \langle \text{L4}, x = 0.5 \rangle \xrightarrow{\text{bright}!} \langle \text{BRIGHT}, x = 0.5 \rangle$ resulting in the observable trace $\sigma = 5 \cdot \text{touch}^? \cdot \text{dim}! \cdot 3.14 \cdot \text{touch}^? \cdot 0.5 \cdot \text{bright}!$. Note that $\{\langle \text{OFF}, x = 0 \rangle\}$ After $\sigma = \{\langle \text{BRIGHT}, x = 0.5 \rangle\}$, $\text{Out}(\{\langle \text{OFF}, x = 0 \rangle\}$ After $\sigma) = \mathbb{R}_{\geq 0}$, but $\text{Out}(\langle \text{L1}, x = 0 \rangle) = \{\text{dim}!, \text{bright}!\} \cup \{0..2\}$.

The TA shown in Figure 1a and Figure 1b respectively can be composed in parallel on actions $A_{in} = \{\text{touch}\}$ and $A_{out} = \{\text{off}, \text{dim}, \text{bright}\}$ forming a closed network.

Using the UPPAAL tool it is possible to edit, simulate and model-check properties of UPPAAL TA in a graphical environment. The property specification language supports safety, liveness, deadlock, and response properties. The UPPAAL tool performs symbolic reachability analysis of the network of TA to search for reachable states where the property is satisfied (or not satisfied). If a state satisfying the property is found, UPPAAL generate a diagnostic trace witnessing the property.

UPPAAL uses a variety of compact data structures for representing the dense state-space, e.g., by *difference bounded matrix* [14] that can be efficiently manipulated by constraint-solving techniques [25], implemented as model-checking tools such as UPPAAL and Kronos [13].

2.3 Timed I/O Game Automaton

To characterize the uncontrollability of some actions, we adopt the notion of Timed Game Automata. A *timed game automaton* (TGA) [23] is a timed automaton with its set of edges E partitioned into controllable ones E_c and uncontrollable ones E_u .

In this paper we further refine the above definition by assuming all output actions A_{out} to be uncontrollable and all input actions A_{in} to be controllable. Thus, a *timed I/O game automaton* (TIOGA) is a timed game automaton where only edges with input actions are controllable, i.e., $(\ell, g, a, u, \ell') \in E_c$ iff $a \in A_{in}$.

The tester acts as a player being in control of the controllable actions and the IUT acts as the opponent player choosing among the uncontrollable ones. A *run* of the TIOGA involves a sequence of tester-proposed input stimuli and actual IUT-produced reactions.

Figure 1a is a TIOGA of the light controller. Solid lines represent transitions of controllable actions and dotted lines represent transitions of uncontrollable actions. Note that this specification exhibits both *output uncontrollability* where a state has multiple output actions enabled ($\langle \text{L1}, x = 0 \rangle \xrightarrow{\text{dim}!}$, and $\langle \text{L1}, x = 0 \rangle \xrightarrow{\text{bright}!}$), and *timing uncertainty of outputs* where an output once enabled is not required to be produced immediately, i.e., both time may elapse and an output occur ($\langle \text{L4}, x = 0 \rangle \xrightarrow{\text{bright}!}$, and $\langle \text{L4}, x = 0 \rangle \xrightarrow{d}, d \in \mathbb{R}_{\geq 0}$). Hence, the user/tester does not know when or which output will be produced. Hence, a test case in general need to branch out and be adaptive to uncontrollable responses. In the untimed finite-state case a common technique

is to use a sub-set construction to determinize the specification, and from this unfold test trees. However, for timed automata, determinization is in general uncomputable. Moreover, branching for each time tick will create a very large tree. Finally, observe in Figure 4 how the state-set reachable after a given observable trace captures state-uncertainty: $\langle \text{Off}, x = 0 \rangle \text{After } 2 \cdot \text{touch?} \cdot 3.5 = \{\langle \text{Dim1}, x \rangle | x \leq 3.5\} \cup \{\langle \text{Dim2}, x \rangle | x \leq 3.5\} \cup \{\langle \text{Bright}, x \rangle | x \leq 3.5\} \cup \{\langle \text{Off}, x \rangle | x \leq 0.5\}$.

2.4 Relativized Timed Conformance

In this section we define our notion of conformance between TIOTSs. Our notion derives from the widely used input/output conformance relation (*ioco*) of Tretmans [27,30] by taking time and environment constraints into account.

Whereas *ioco* used quiescence to capture the absence of outputs, now and in the (unbounded) future, in a real-time relation, absence of outputs is always for a particular period of time, which is just observing passing of time without an action occurring. This means that in timed testing the concept of quiescence is not necessary.

Under assumptions of weak input enabledness our relativized timed conformance relation (denoted rtioco_e) coincides with relativized timed trace inclusion. Like *ioco*, this relation ensures that the implementation has only the behavior allowed by the specification. In particular, 1) it is not allowed to produce an output at a time when one is not allowed by the specification, 2) it is not allowed to omit producing an output when one is required by the specification.

Definition of rtioco_e . Let $\mathcal{S} = (S, s_0, A_{in}, A_{out}, \rightarrow)$ be a weak-input enabled and non-blocking TIOTS. An *environment* for \mathcal{S} is itself a weak-input enabled and non-blocking TIOTS $\mathcal{E} = (E, e_0, A_{out}, A_{in}, \rightarrow)$ with reversed inputs and outputs. Given an environment state $e \in E$ the e -relativized timed input/output conformance relation rtioco_e between system states $s, t \in S$ is defined as:

$$s \text{rtioco}_e t \text{ iff } \forall \sigma \in \text{TTr}(e). \text{Out}((s, e) \text{After } \sigma) \subseteq \text{Out}((t, e) \text{After } \sigma)$$

Whenever $s \text{rtioco}_e t$ we will say that s is a correct implementation (or refinement) of the specification t under the environmental constraints expressed by e .

For example, an IUT \mathcal{J} that produced output *bright!* after a touch would not be conforming to the specification \mathcal{S}' in Figure 1 because $\{\text{bright!}\} \not\subseteq \text{Out}(\langle \text{OFF}, x = 0 \rangle \text{After } \text{touch?}) = \{\text{dim!}\} \cup \{0..2\}$. Similarly, an \mathcal{J}' that produced a late dim after 3 time units is also non-conforming because $\{0..3\} \not\subseteq \text{Out}(\langle \text{OFF}, x = 0 \rangle \text{After } \text{touch?}) = \{\text{dim!}\} \cup \{0..2\}$.

Under the assumption of weak input-enabledness of both \mathcal{S} and \mathcal{E} we may characterize relativized conformance in terms of trace-inclusion as follows:

Lemma 1. *Let \mathcal{S} and \mathcal{E} be input-enabled with states $s, t \in S$ and $e \in E$ resp., then*

$$s \text{rtioco}_e t \text{ iff } \text{TTr}(s) \cap \text{TTr}(e) \subseteq \text{TTr}(t) \cap \text{TTr}(e)$$

Thus if $s \text{rtioco}_e t$ does not hold then there exists a trace σ of e such that $s \xrightarrow{\sigma}$ but $t \not\xrightarrow{\sigma}$. E.g, in the above example, $\text{touch?} \cdot \text{bright!} \in \text{TTr}(\mathcal{J})$, but not in $\text{TTr}(\mathcal{S}')$, and $\text{touch?} \cdot 3 \cdot \text{dim!} \in \text{TTr}(\mathcal{J}')$, but not in $\text{TTr}(\mathcal{S}')$.

In particular there is a most discriminating environment U that can generate all traces over A : $\text{TTr}(U) = (A \cup \mathbb{R}_{\geq 0})^*$. The corresponding conformance relation rtioco_U then specializes to simple timed trace inclusion (timed output trace inclusion) between system states.

Test Purposes. A *test purpose* is a specific observation objective that the tester would like the IUT to produce. For each system requirement the tester formulates a set of such test purposes. In model-based testing, test purposes are typically specified as message sequence charts, observer automata, or as logical properties. Here we simply assume that the purpose specifies a set of possible goal states \mathcal{P} of the model that the test run should *visit*. A test purpose could e.g. be to check that the light level in Figure 1 can become bright. In UPPAAL, \mathcal{P} can be characterized by the set of states satisfying the UPPAAL TCTL property $E\langle \text{IUT.Bright} \rangle$.

3 Timed Test Generation

We present a number of principles and approaches for generating timed test cases for TA models with varying restrictions.

3.1 Testing Deterministic Controllable TA

We start by introducing a fully controllable class of TA where test generation is easy, but without any uncertainty. This defines the base class that we try to generalize. The idea is to formulate the test case generation problem as a reachability problem, which can be solved with an existing model-checking tool.

In order to make this approach to offline test case generation applicable to TA specifications, we shall assume that the underlying TIOTS is *deterministic, weakly input enabled, output urgent, with isolated outputs* as defined in Section 2.1. We term this class of TA DOUTA. This means that \mathcal{S} is assumed to react deterministically to any input provided, and will always be able to accept input from the test case. At any state, the \mathcal{S} is also assumed to always have at most one output action that will occur immediately.

A DOUTA version of the light controller is shown in Figure 2. Note the use of committed locations that forces the TA to issue its (single) output immediately. In DOUTA the tester has thus full control over which states are visited with a given input sequence. This means that a test case is generated as simple sequence of actions from a model checker such that this trace is in the specification and that it satisfies the test purpose, see Algorithm 1. Thus, a *test sequence* is an alternating sequence of concrete delay actions and observable actions.

Discussion. The generated test sequences are valid because they are derived from diagnostic traces, and are thus guaranteed to be included in the specification. It is also possible to use this technique to generate test sequences with a guaranteed coverage of the model by formulating coverage as a reachability question as explained in [16] and as implemented in UPPAAL-COVER, or that are as short, as fast, or as cheap (using priced TA) as possible to execute. In conclusion, if the IUT falls into this class, efficient and simple test generation techniques exist based on model-checking.

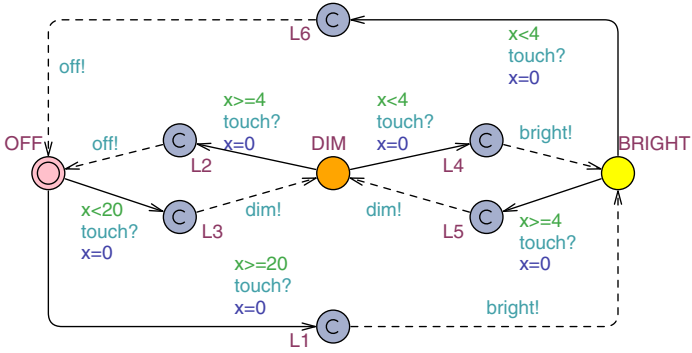


Fig. 2. Simple Light Controller DOUTA

Algorithm 1. From Diagnostic Traces to Test Cases.

input: $\mathcal{M} = \mathcal{S} \parallel \mathcal{E}$ composed of IUT model \mathcal{S} and environment \mathcal{E} ; test purpose \mathcal{P} .

1. A diagnostic trace produced by UPPAAL for a given reachability question for \mathcal{P} on \mathcal{M} demonstrates the sequence of moves to be made by each of the system components and the required clock constraints needed to reach the target location. A (concrete) diagnostic trace has the form:

$$(s_0, e_0) \xrightarrow{\gamma_0} (s_1, e_1) \xrightarrow{\gamma_1} (s_2, e_2) \xrightarrow{\gamma_2} \dots (s_n, e_n)$$

where s_i, e_i are states of the \mathcal{S} and \mathcal{E} , respectively, and γ_i are either time-delays or synchronization (or internal) actions. The latter may be further partitioned into purely \mathcal{S} or \mathcal{E} transitions (hence invisible for the other part) or synchronizing transitions between \mathcal{S} and \mathcal{E} (hence observable for both parties).

2. A test sequence, $\lambda \in (A_{in} \cup A_{out} \cup \mathbb{R}_{\geq 0})^*$, may be obtained simply by projecting the diagnostic trace to the \mathcal{E} -component, while removing invisible transitions, and summing adjacent delay actions.
3. The application of the test sequence $\sigma = apply(\lambda, IUT)$ consists of executing it in parallel with the IUT synchronizing over observable actions. Remarking that a test sequence and the IUT both can be understood as TIOTS, the application can be formalized as a maximal run of $\lambda \parallel IUT$ from which the resulting observable trace can be extracted.
4. A PASS verdict must only be issued if the execution match the entire test sequence:

$$verdict = \begin{cases} \text{PASS} & \text{if } apply(\lambda, IUT) = \lambda \\ \text{FAIL} & \text{otherwise} \end{cases}$$

3.2 Preset Input Sequences

The DOUTA assumption is frequently impractical or unrealistic for embedded real-time systems. It allows no uncertainty whatsoever, and no internal actions or synchronizations (except in very special cases).

However, offline test generation using a real-time model-checking engine for unrestricted TA models is still possible using the procedure in Algorithm 2 that separates test input generation and test evaluation in two different steps. The basic idea is to

Algorithm 2. Testing using preset input sequences

Input: Model $\mathcal{M} = \mathcal{S} \parallel \mathcal{E}$, test purpose \mathcal{P} , implementation IUT

1. Generate a preset timed input sequence by using the model-checker to generate a concrete diagnostic trace satisfying the test purpose. Project it on the \mathcal{E} -component while *removing outputs*, internal actions, and summing up adjacent delays. Let λ_i denote this input trace.
2. Execute the preset-timed input sequence on the IUT, and record the resulting timed input/output sequence: $\sigma = \text{apply}(\lambda_i, \text{IUT})$
3. Check whether the observed trace is included in the specification model and whether the test purpose is (possibly) satisfied:

$$\text{verdict} = \begin{cases} \text{PASS} & \text{if } \sigma \in \text{TTr}(\mathcal{M}) \text{ and } \exists \rho \in \text{Runs}(\mathcal{S} \parallel \sigma). \text{Visits}(\rho) \cap \mathcal{P} \neq \emptyset \\ \text{INCONC} & \text{if } \sigma \in \text{TTr}(\mathcal{M}) \text{ and } \forall \rho \in \text{Runs}(\mathcal{S} \parallel \sigma). \text{Visits}(\rho) \cap \mathcal{P} = \emptyset \\ \text{FAIL} & \text{if } \sigma \notin \text{TTr}(\mathcal{M}) \end{cases}$$

compute an preset input sequence that satisfies the test purpose, then apply this sequence on the IUT, and finally evaluate whether the resulting behavior is conforming.

Observe that we have two kinds of pass-verdicts: PASS and INCONC. An INCONC verdict is generally issued when the purpose of a test cannot be demonstrated, but also no evidence of non-conformance is found. When a specification (or implementation) is non-deterministic the implementation may produce a different (legal) output sequence than what was expected to satisfy the test purpose.

To check whether an observed trace is a trace of the model is fairly simple: put the observed trace (or a TA model thereof) in parallel with the specification model instead of its environment model, and check that the trace can execute entirely. In a similar way it can be checked whether the test purpose (possibly or always) holds¹ in the states reachable after the observed trace.

Discussion. The above procedure is practical, computationally efficient, and operational. However, a particular disadvantage is that it cannot guarantee a-priori that the test purpose will be satisfied on a correct implementation due to the non-adaptiveness of the preset input sequence. One can try to re-execute the input sequence, but in the worst case the implementation will never reveal the test purpose given the (arbitrarily) chosen input sequence.

3.3 Online Testing

An online testing tool executes test events interactively as they are generated by performing a guided random exploration of the environment model while checking system outputs against the system model. The main idea is continually to compute the possible set of states S the (combined system and environment) model can occupy as test inputs,

¹ In the presented definition we only require that it is possible to satisfy the test purpose under some execution of the model on the observed trace. It is also possible to check the stricter statement that the test purpose must be satisfied under all executions of the model on the observed trace.

outputs or delays are observed. The state-set captures the uncertainty that the tester has of the possible state a (conforming) IUT may be in.

UPPAAL-TRON [16,21,20] is an online test generation tool that is integrated with the UPPAAL model-checker, and uses the same TA syntax and semantics. The algorithm behind UPPAAL-TRON is shown in Algorithm 3. It uses the state-set to compute the possible set of inputs that the tester can offer, and the set of allowed system outputs. The main loop runs until the testing time is over, or non-conformance is detected. It randomly chooses between offering an input to the IUT, waiting and observing the IUT for outputs, or re-starting. If an output occurs, it checks if this is legal in the current state-set; else it declares non-conformance. After each input, output or delay action the set of states S is updated.

Algorithm 3. Test generation and execution, $\text{OnlineTest}(S, \mathcal{E}, \mathcal{P}, \text{IUT}, \mathcal{J})$.

```

1  $S := \{\langle s_0, e_0 \rangle\};$  // let the set contain an initial state
2 if computation of After in any step below reach a state  $s \in \mathcal{P}$  then return PASS
3 while  $S \neq \emptyset \wedge \#iterations \leq To$  do
4   switch  $\text{Random}(\{action, delay, restart\})$  do
5     case action // offer an input
6       if  $\text{EnvOutput}(S) \neq \emptyset$  then
7         randomly choose  $i \in \text{EnvOutput}(S)$ ;
8         send  $i$  to IUT,  $S := S \text{ After } i$ ;
9     case delay // wait for an output
10      randomly choose  $d \in \text{Delays}(S)$ ;
11      sleep for  $d$  time units or wake up on output  $o$  at  $d' \leq d$ ;
12      if  $o$  occurs then
13         $S := S \text{ After } d'$ ;
14        if  $o \notin \text{ImpOutput}(S)$  then return FAIL;
15        else  $S := S \text{ After } o$ 
16      else  $S := S \text{ After } d$ ; // no output within  $d$  delay
17     case restart // reset and restart
18        $S := \{\langle s_0, e_0 \rangle\}$ ;
19       reset IUT
20 if  $S = \emptyset$  then return FAIL else return INCONC

```

Discussion. An advantage of online testing is that it allows very expressive models (e.g., the full UPPAAL TA model without restrictions on non-determinism) as long as the state set can be computed and analyzed wrt. inputs and outputs. Further, it is automatically adaptive to the actual outputs (and their timing), and will generate relevant stimuli in response.

A potential disadvantage is that the tool must compute the state set in real-time (by performing symbolic exploration). However, UPPAAL-TRON does this quite efficiently by using and extending the compact symbolic data-structures and state exploration algorithms implemented in the UPPAAL engine. Case studies have shown that it can manage

large models² with a high degree of uncertainty. It is thus feasible in practice for many real-time systems with time constraints down to the milli-second range.

Another disadvantage is the randomization; satisfaction of coverage criteria or test purposes cannot generally be guaranteed apriorily.

3.4 Observable Timed Automata Using Timed Games

The previous sections treated test generation as a reachability problem or as a randomized execution. However, when the goal is to satisfy a given test purpose for systems that are uncontrollable a better approach for test generation may be using a two-player reachability *game*. Here the tester is in control of which input to offer, and tries to satisfy the test purpose. The IUT is in control of which output to deliver, and behaves like an adversary trying to prevent the tester from meeting his objective. The system and environment models specify the possible moves of the tester and (conforming) implementations. A winning strategy is a function that gives the environment (tester) step-by-step guidance on what actions to take, and that ensures that no matter how “bad” the IUT behaves, the test purpose will be satisfied. The idea is to use a winning strategy as test case and use a timed game solver to generate tests.

An *observable* TIOGA is deterministic and input enabled such that any observable trace leads to a unique system state: $\forall \sigma \in TTr(\mathcal{E}). |\mathcal{S} \text{ After } \sigma| = 1$. But timing uncertainty and output action uncertainty is specifically allowed, hence it is not fully controllable. Figure 1 is an example of an *observable* TIOGA.

The framework of testing with winning strategies is illustrated in Figure 3. The inputs to UPPAAL-TIGA are the game models of IUT, its environment, and the test purpose in a formula of a subset of the ACTL logic. The output is a winning strategy, when the property is satisfied.

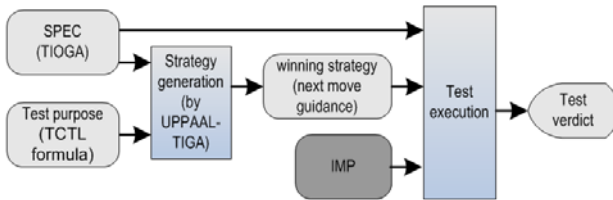


Fig. 3. Testing with winning strategies

We view the reachability control problem $(\mathcal{S}, \mathcal{P})$ as a game problem to compute a winning strategy λ such that \mathcal{S} supervised by λ can reach some states in \mathcal{P} [23].

A finite or infinite run ρ of \mathcal{S} is *winning* if $Visits(\rho) \cap \mathcal{P} \neq \emptyset$. The set of all winning runs in \mathcal{S} starting from state s is denoted by $WinRuns(s, \mathcal{P})$. In the following let \mathcal{S} be a M-bounded TIOGA, and let $(\mathcal{S}, s_0, A_{in}, A_{out}, \rightarrow)$ be the TIOTS of \mathcal{S} . A *state-based strategy* λ over \mathcal{S} is a partial function from \mathcal{S} to $A_{in} \cup \{\text{delay}\}$, where A_{in} is the controllable

² E.g., the Danfoss case [21] used models with 18 concurrent automata, 14 clocks and 14 integer variables.

input actions, and delay signifies the idle action where the player does nothing except wait. If a player plays the game always according to what a strategy λ suggests, the resulting run is called a λ supervised run, i.e. $\text{SupRuns}(s, \lambda) \subseteq \text{Runs}(s, \mathcal{S})$.

λ is *winning* from state s if $\text{MaxRuns}(s) \cap \text{SupRuns}(s, \lambda) \subseteq \text{WinRuns}(s, \mathcal{P})$. If λ is winning from s_0 , then λ is called a *winning strategy* for \mathcal{S} .

UPPAAL-TIGA is a timed game solver, which can check whether a specified ACTL test purpose can be satisfied by a TGA, and if so, automatically synthesize a winning strategy; note that there may exist more than one winning strategy for the same TGA and test purpose. The strategy function may be output as a set of condition-action statements of the form: “if the IUT is in states $\langle \text{Dim}, 0 \leq x < 4 \rangle$, the tester should offer a *touch*?; if in states $\langle \text{L1}, x \leq 2 \rangle$, the tester should just wait”.

The basic idea of test execution is to continuously consult the winning strategy and the SPEC model, as shown in Algorithm 4. If an occurred output is prohibited according to rtico, we report FAIL, otherwise after reaching a goal state we report PASS.

Algorithm 4. Test Execution of a Winning Strategy

```

1 Input: TIOGA specification  $\mathcal{M}$ , system implementation IUT, goal states  $\mathcal{P}$ , and
   state-based winning strategy  $\lambda$ ;
2 Output: test verdict PASS or FAIL, and observable trace  $\sigma$ ;
3  $\sigma := \langle \rangle$ ; // the test run is initially an empty trace;
4 while ( $\sigma \notin \text{WinRuns}(s_0, \mathcal{S}, \mathcal{P})$ ) do //  $s_0$ : init state
5   switch  $\lambda(S \text{ After } \sigma)$  do // Consult strategy
6   | case input  $i$ 
7   |   send  $i$  to IUT;
8   |    $\sigma := \sigma \cdot i$ ;
9   | case delay  $d$ 
10  |   if output  $o$  occurs at  $d' \leq d$  then
11  |   |    $\sigma := \sigma \cdot d'$ ;
12  |   |   if  $o \notin \text{Out}(s_0 \text{ After } \sigma)$  then return FAIL;
13  |   |   else  $\sigma := \sigma \cdot o$ ;
14  |   | else
15  |   |   if  $d \notin \text{Out}(s_0 \text{ After } \sigma)$  then return FAIL;
16  |   |   else  $\sigma := \sigma \cdot d$ ;
17 return PASS

```

Discussion. The advantage of this approach (further details in [11][22]) is that it enables test generation for a much more general class of systems than DOUTA, and still guaranteeing observation of the test purpose (on conforming IUT’s when a winning strategy exists; on non-conforming IUT’s guarantees can obviously never be given). If a winning strategy does not exist, the test purpose may be reformulated. Alternatively, we have shown how to generate *cooperative strategies* [10] where the tester first executes in a randomized online fashion within a restricted state space until he encounters a state from which a winning strategy exists.

Still the class is restrictive, i.e. it does not allow internal non-determinism (that e.g. might arise from internal synchronization on parallel composed specification components) and assumes exact observations. Also, solving timed games may be computationally expensive (in particular cooperative strategies), however, our experiments [11] indicate that it may indeed be feasible in many realistic cases.

3.5 Testing under Partial Observability

Section 3.4 considered testing under full observability. However, many systems are only partially observable, e.g. where the observer/tester cannot measure exactly the value of output variables or shared clocks, or cannot immediately see the effects of internal actions. Thus, the observer cannot infer the exact state of the IUT, and the method in Section 3.4 is inapplicable. In this section we relax this restriction and consider testing timed systems that are only *partially observable* (or, with *imperfect information*).

We characterize partial observability in terms of a finite number of possible observations to be made on the IUT states [9]. Consider the “flickering” light controller in Figure 4a, which have four brightness levels (in ascending order): Off, Dim1, Dim2 and Bright. Initially, it is in location Off. If the pad is touched at an appropriate time ($x \geq 2$), the light will go to location L1 where within 2 time units it will non-deterministically go to Bright or go to Dim1. At location Dim1, a *touch?* input at appropriate time can bring the TA to Dim2. The light can automatically go to Dim2 at any time. If clock x rises up to 3 while the light is in Dim1, then it can automatically go Off. In Figure 4a, the edges that are not labeled by *touch?* are internal transitions, which need no synchronization with the user TA in Figure 4.

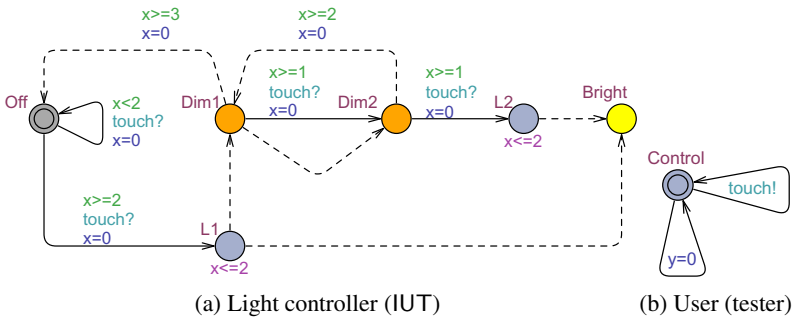


Fig. 4. Flickering light controller example

Note that in Dim1, Dim2, L1 and L2, internal transitions can autonomously occur when the conditions (if any) are satisfied, and their occurrences cannot be observed by the user. Further, the light itself decides whether or not to make an output or an internal transition, and if yes, which output or internal transition to make at what time. Thus, this example has all of *internal actions*, *uncontrollable actions* and *timing uncertainty of uncontrollable actions*.

In this example we assume that the tester can observe³:

- (1) whether or not the IUT is “off” (i.e., in Off); and
- (2) whether or not the IUT is “dim” (i.e., in Dim1 or Dim2, but not exactly in which one); and
- (3) whether or not the IUT is “bright” (i.e., in Bright),

Consider that the tester feeds the IUT with a timed input sequence $2 \cdot \text{touch?} \cdot 2$ at the initial state of Figure 4a. He may get the observation “dim” signaling that either location Dim1 or location Dim2 has been reached, or the observation “bright”.

Observations. Let $A = (L, l_0, A, X, E, I)$ be an M-bounded TGA. Let $K \subseteq L$ and $\varphi \in \mathcal{G}(X)$. We call (K, φ) an *observable predicate*. We use a finite set of observable predicates $P \subseteq 2^L \times \mathcal{G}(X)$ to observe a TGA. For instance, in Figure 4a we can have $P = \{(\{\text{Off}\}, \text{true}), (\{\text{Dim1}, \text{Dim2}\}, \text{true}), (\{\text{Bright}\}, \text{true}), (L, 0 \leq y < 1)\}$ ⁴.

An observable predicate (K, φ) is true at a TGA semantic state $s = \langle l, \bar{v} \rangle$ iff $l \in K$ and $\bar{v} \models \varphi$. A *state observation* (or *observation* for short) $o_{P,s}$ of the TGA with a set P of predicates at state s is a valuation of all the predicates in P at s . E.g., in Figure 4a at semantic state $\langle \text{Dim2}, (x = 2, y = 4) \rangle$ we have the observation $o_{P,s}$ such that

$$\begin{aligned} o_{P,s}(\{\{\text{Off}\}, \text{true}\}) &= \text{false}, \\ o_{P,s}(\{\{\text{Dim1}, \text{Dim2}\}, \text{true}\}) &= \text{true}, \\ o_{P,s}(\{\{\text{Bright}\}, \text{true}\}) &= \text{false}, \text{ and} \\ o_{P,s}(L, 0 \leq y < 1) &= \text{false}. \end{aligned}$$

Let O_P be the set of all possible observations with P , then by definition $|O_P| \leq 2^{|P|}$. During a *play* the environment (tester) may propose a specific controllable action or to delay $c_i \in (A_{in} \cup \{\text{delay}\})$. The opponent (IUT) may however prevent this (only) if the action is not enabled, or by issuing an output itself. A play is a run of actually occurring actions $\alpha_i \in (A_{in} \cup A_{out} \cup \{\tau\} \cup \mathbb{R}_{\geq 0})$. If a play ρ is a finite sequence, then it is called a *prefix*. The set of all prefixes of a TGA is denoted as Pref. A *strategy* for a partially observable TGA is a function $\lambda : \text{Pref} \rightarrow (A_{in} \cup \{\text{delay}\})$ instructing the tester, given the observation history, what next action to take. Thus it is a trace-based (history-based) strategy. In a *stuttering-invariant strategy* repeated identical observations does not change the strategy. The player “sticks to”, or repeats, the suggested controllable move or delay until the observations change.

Under a set P of observable predicates, an *observation history* is a function $\text{Obs}_P : \text{Pref} \rightarrow O_P^*$, which maps a prefix $\rho \in \text{Pref}$ to the chronological sequence $\text{Obs}_P(\rho)$ of non-stuttering observations along ρ .

³ In practice, the observable predicates of (1)-(3) can be implemented by probing/instrumenting the IUT with some light sensors or software-defined location reporters; the assumption in predicate (2) is reasonable if the difference between these two brightness levels is too small to be discerned by the light sensors.

⁴ The constraint $0 \leq y < 1$ means that the tester can check whether value of clock y falls within $[0, 1)$, but he cannot/need not read the exact value of y . In practice, this predicate thus can be implemented as a countdown timer whose timeout can be externally observed by the tester.

Game Solving. UPPAAL-PO-TIGA is a timed game solver for partially observable timed games. Given a network of bounded TGA models of the system and its environment, a set of observable predicates, and a winning objective in terms of an ACTL reachability or safety property, it model-checks whether the property can be enforced on the models, and if the outcome is positive, it extracts an *observation-based stuttering-invariant winning strategy* from the explored paths.

The algorithm uses a knowledge-based subset construction and on-the-fly partially observable reachability (OTFPOR) computation [9], which is based on a mixture of forward search and backward propagation timed game solving algorithm [8]. A key point is that the algorithm uses the observable predicates to partition the state-space.

For a reachability property φ , a winning strategy can be represented as a directed acyclic graph which has an initial node corresponding to the initial state, and a number of leaf nodes corresponding to the states that satisfy φ .

Figure 5 shows a winning strategy λ_R that is generated by UPPAAL-PO-TIGA for the example in Figure 4 and the reachability property “control : $A \diamond$ Bright”. Each node in Figure 5 corresponds to an observation history, and each of its outgoing edges correspond to observation changing application of the strategy action. We list only those observable predicates that evaluate to true under that observation.

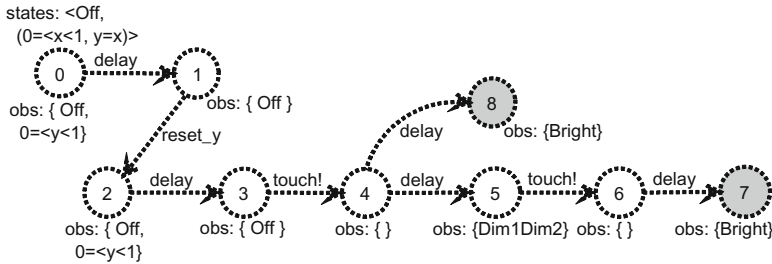


Fig. 5. Winning strategy λ_R for reachability property control : $A \diamond$ Bright

In Figure 4a, if we consider the following prefix ρ :

$$\langle \text{Off}, (x=0, y=0) \rangle \cdot \text{delay} \cdot 1 \cdot \langle \text{Off}, (x=1, y=1) \rangle \cdot \text{reset}_y \cdot \text{reset}_y \cdot \langle \text{Off}, (x=1, y=0) \rangle \cdot \text{delay} \cdot 1 \cdot \langle \text{Off}, (x=2, y=1) \rangle \cdot \text{touch} \cdot \text{touch} \cdot \langle \text{L1}, (x=0, y=1) \rangle \cdot \text{delay} \cdot 0 \cdot \langle \text{Dim1}, (x=0, y=1) \rangle \cdot \text{touch} \cdot 1 \cdot \langle \text{Dim1}, (x=1, y=2) \rangle \cdot \text{touch} \cdot \text{touch} \cdot \langle \text{Dim2}, (x=0, y=2) \rangle \cdot \text{touch} \cdot 1 \cdot \langle \text{Dim2}, (x=1, y=3) \rangle \cdot \text{touch} \cdot \text{touch} \cdot \langle \text{L2}, (x=0, y=3) \rangle,$$

then the observation history of ρ is:

$$\{\text{Off}, 0 \leq y < 1\} \cdot \{\text{Off}\} \cdot \{\text{Off}, 0 \leq y < 1\} \cdot \{\text{Off}\} \cdot \{\} \cdot \{\text{Dim1Dim2}\} \cdot \{\}.$$

The above history corresponds to the trace 0-1-2-3-4-5-6 in Figure 5. Note that there may exist choices in a strategy, i.e., there might exist more than one possible next observation. For instance, in Figure 5, where the tester sticks to the *delay* move at node #4, there may be a new observation of either that in node #5 or that in node #8. This branching of the strategy is due to the reaction uncertainties in the TGA models.

Test Execution. The test execution algorithm for partially observable strategies resembles Algorithm 4 except in two main ways. It must continuously evaluate the observable predicates and track the observation history to consult the strategy. Secondly, it must declare FAIL if it makes an observation o_{P,s_i} of the implementation in some state s_i that cannot be made on the specification (in the same spirit of $rtioco_E$), and declare PASS if a winning state is reached.

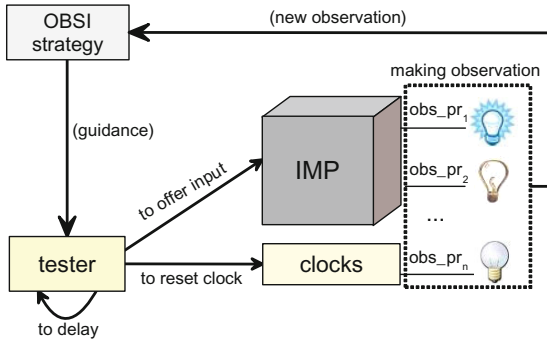


Fig. 6. Observation-based conformance testing

Figure 6 is a schematic view of observation-based conformance testing of *partially observable* timed systems. It must be possible for the tester to determine the truth of each observable predicate from the observable output actions of the implementation, from its sensors, from externally observable output variables or clocks, or from “probes” into the IUT (a.k.a. gray box view) that reports whether the values of system variables are within some particular intervals. These are generally fairly realistic assumptions.

Discussion. Partial observability enables test generation with guaranteed satisfied test purposes for a rich and practical class of TA and with realistic imperfect observations. In the case that a winning strategy does not exist for the test purpose, the tester can be given more observational power, corresponding to additional instrumentation and correspondingly more observable predicates in the game problem. Details of the approach is given in [12]. Surprisingly, our experiments [12] show that the computation resources needed to solve partially observable games in many cases are significantly smaller than when using fully observable games; the reason, however, is that the state-space partitioning resulting from the observable predicates typically are coarser than when using full observability. Thus this is a promising approach.

4 Related Work

The work here surveys [16,20,21,11,10,12] with the addition of Section 3.2. Timed automaton (TA) [2] has elsewhere been widely used to model real-time systems. A considerable proportion of existing efforts on real-time testing [15,26,17,24,16,7,18,19]

are based on the TA model or its variants. Among them some make the assumptions that the system model is output-urgent and has isolated outputs [15,26,24]. In contrast, [19] presents both an online testing of non-deterministic TA in a similar style as ours, and also an offline algorithm for non-deterministic systems by explicitly modeling the tester's time observation capabilities through a digital clock. But unlike our game based approach this does not guarantee satisfaction of the test purpose after execution. [3] proposes a game based approach for computing an (overapproximate) determinized TA from which test cases can be generated, but offer no implementation.

To enable conformance testing, these methods build their implementation relations (a.k.a. conformance relations) on top of e.g. trace equivalence [15,26] or the iOCO conformance relations [17,20,7,18]. A detailed comparison of different versions of timed conformance is given in [28]. An approximate conformance relation for imprecise systems based on distances of quantitative transition systems is presented in [5]. Reactive planning to guide an online tester towards a test purpose or improved coveragee has been proposed in [29].

Game-theoretic approaches to untimed system testing have been discussed in [1,31,4]. An alternative way to characterize partial observability is to assume that only a proper subset of those outputs from the IUT, and/or only a proper subset of the system clocks can be read by the tester [6].

5 Conclusions

Effective methods for testing of embedded real-time systems need to embrace state and observation uncertainty, and imperfect control and observation. We have outlined a number of test generation principles and algorithms, each is supported by practical tools in the UPPAAL family.

The approaches differ in the extend to which they allow uncertainty, their ability to apriori guarantee satisfaction of the test purpose, and in the required amount of offline- and online computation resources. It is not straightforward to determine from the syntactic declarations of the model what online and offline computation resources each method requires and how they scale, as this depends greatly on the behavior of the underlying transition system and state space. We view the methods presented here as complementary.

When systems are essentially controllable a good technique is to use diagnostic traces. If the system is highly non-deterministic, or the model is very large, online testing is effective. For moderate sized systems where the test purpose must be satisfied, or when very fast execution resolution is required the offline approaches are preferable. In particular we find testing based on partially observable games an interesting and general technique that we will explore further in future work.

References

1. Alur, R., Courcoubetis, C., Yannakakis, M.: Distinguishing tests for nondeterministic and probabilistic machines. In: Proc. STOC 1995, pp. 363–372. ACM Press, New York (1995)
2. Alur, R., Dill, D.L.: A theory of timed automata. *Theoretical Computer Science* 126(2), 183–235 (1994)

3. Bertrand, N., Jéron, T., Stainer, A., Krichen, M.: Off-line test selection with test purposes for non-deterministic timed automata. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 96–111. Springer, Heidelberg (2011)
4. Blass, A., Gurevich, Y., Nachmanson, L., Veanes, M.: Play to test. In: Grieskamp, W., Weise, C. (eds.) FATES 2005. LNCS, vol. 3997, pp. 32–46. Springer, Heidelberg (2006)
5. Bohnenkamp, H., Stoelinga, M.: Quantitative testing. In: Proc. EMSOFT 2008. ACM, New York (2008)
6. Bouyer, P., D'Souza, D., Madhusudan, P., Petit, A.: Timed control with partial observability. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 180–192. Springer, Heidelberg (2003)
7. Briones, L.B., Brinksma, E.: A test generation framework for *quiescent* real-time systems. In: Grabowski, J., Nielsen, B. (eds.) FATES 2004. LNCS, vol. 3395, pp. 64–78. Springer, Heidelberg (2005)
8. 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)
9. 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)
10. David, A., Larsen, K.G., Li, S., Nielsen, B.: Cooperative testing of timed systems. In: Proc. MBT 2008 (2008)
11. David, A., Larsen, K.G., Li, S., Nielsen, B.: A game-theoretic approach to real-time system testing. In: Proc. DATE 2008 (2008)
12. David, A., Larsen, K.G., Li, S., Nielsen, B.: Timed testing under partial observability. In: Proc. 2nd International Conference on Software Testing, Verification and Validation (ICST 2009), Denver, Colorado, USA, pp. 61–70. IEEE Computer Society, Los Alamitos (2009)
13. Daws, C., Olivero, A., Yovine, S.: Verifying ET-LOTOS programs with KRONOS. In: Hogrefe, D., Leue, S. (eds.) Proc. of 7th Int. Conf. on Formal Description Techniques. North-Holland, Amsterdam (1994)
14. Dill, D.: Timing Assumptions and Verification of Finite-State Concurrent Systems. In: Sifakis, J. (ed.) CAV 1989. LNCS, vol. 407, pp. 197–212. Springer, Heidelberg (1990)
15. En-Nouaary, A., Dssouli, R., Khendek, F., Elqortobi, A.: Timed test cases generation based on state characterization technique. In: Proc. RTSS 1998, pp. 220–229 (1998)
16. Hessel, A., Mikucionis, M., Nielsen, B., Pettersson, P., Skou, A., Larsen, K.G.: Testing real-time systems using UPPAAL. In: Hierons, R.M., Bowen, J.P., Harman, M. (eds.) FORTEST. LNCS, vol. 4949, pp. 77–117. Springer, Heidelberg (2008)
17. Khoumsi, A., Jéron, T., Marchand, H.: Test cases generation for nondeterministic real-time systems. In: Petrenko, A., Ulrich, A. (eds.) FATES 2003. LNCS, vol. 2931, pp. 131–146. Springer, Heidelberg (2004)
18. Krichen, M., Tripakis, S.: Black-box conformance testing for real-time systems. In: Graf, S., Mounier, L. (eds.) SPIN 2004. LNCS, vol. 2989, pp. 109–126. Springer, Heidelberg (2004)
19. Krichen, M., Tripakis, S.: Conformance testing for real-time systems. *Formal Methods in System Design* 34(3), 238–304 (2009)
20. Larsen, K.G., Mikucionis, M., Nielsen, B.: Online testing of real-time systems using UPPAAL. In: Grabowski, J., Nielsen, B. (eds.) FATES 2004. LNCS, vol. 3395, pp. 79–94. Springer, Heidelberg (2005)
21. 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, New York (2005)
22. Li, S.: Games and Scenarios for Real-Time System Validation. PhD thesis, Dept. of Computer Science, Aalborg University (2010)

23. Maler, O., Pnueli, A., Sifakis, J.: On the synthesis of discrete controllers for timed systems. In: Mayr, E.W., Puech, C. (eds.) STACS 1995. LNCS, vol. 900, pp. 229–242. Springer, Heidelberg (1995)
24. Nielsen, B., Skou, A.: Automated test generation from timed automata. *STTT* 5(1), 59–77 (2003)
25. Rokicki, T.G., Myers, C.J.: Automatic verification of timed circuits. In: Dill, D.L. (ed.) CAV 1994. LNCS, vol. 818, pp. 468–480. Springer, Heidelberg (1994)
26. Springintveld, J., Vaandrager, F., D’Argenio, P.R.: Testing timed automata. *TCS* 254(1-2), 225–257 (2001)
27. Tretmans, J.: Testing concurrent systems: A formal approach. In: Baeten, J.C.M., Mauw, S. (eds.) CONCUR 1999. LNCS, vol. 1664, pp. 46–65. Springer, Heidelberg (1999)
28. Tretmans, J., Schmaltz, J.: On conformance testing for timed systems. In: Cassez, F., Jard, C. (eds.) FORMATS 2008. LNCS, vol. 5215, pp. 250–264. Springer, Heidelberg (2008)
29. Vain, J., Raiend, K., Kull, A., Ernits, J.P.: Synthesis of test purpose directed reactive planning tester for nondeterministic systems. In: Kurt Stirewalt, R.E., Egyed, A., Fischer, B. (eds.) ASE, pp. 363–372. ACM, New York (2007)
30. de Vries, R.G., Tretmans, J.: On-the-fly conformance testing using SPIN. *Software Tools for Technology Transfer* 2(4), 382–393 (2000)
31. Yannakakis, M.: Testing, optimization, and games. In: Díaz, J., Karhumäki, J., Lepistö, A., Sannella, D. (eds.) ICALP 2004. LNCS, vol. 3142, pp. 28–45. Springer, Heidelberg (2004)

Model-Checking and Simulation for Stochastic Timed Systems*

Arnd Hartmanns

Saarland University – Computer Science, Saarbrücken, Germany

Abstract. For verification and performance evaluation, system models that can express stochastic as well as real-time behaviour are of increasing importance. Although an integrated stochastic-timed verification procedure is highly desirable, both model-checking and simulation currently fall short of providing a complete, fully automatic verification solution. For model-checking, the problem lies in the extreme expressiveness of such a model, while simulation is limited to stochastic processes and cannot deal with nondeterminism. In this paper, we review the use of stochastic timed automata as an overarching formalism to model stochastic timed systems and present two analysis approaches: Model-checking for the (large) subset corresponding to probabilistic timed automata with deadlines, for which solid implementations are appearing, and simulation, which we have recently shown to be applicable to models that also include spurious nondeterministic choices.

1 Introduction

The increasing use of complex, safety-critical or economically vital systems such as fly-by-wire controllers, automated trading systems, “smart” mobile phones and the Internet, creates a similarly increasing need for the formal verification and evaluation of such systems. In this paper, we focus on *model-based* verification: Given a mathematically precise model of the system under study and a set of properties of interest, we want to verify these properties. For qualitative properties, such as “does the system ever reach a bad state” (*safety*) or “whenever the traffic light is red, will it become green again in the future” (*liveness*), verification results are Boolean: The property is either satisfied (*true*, *tt*) or not (*false*, *ff*), the proposed system implementation is correct or incorrect.

However, qualitative verification is more and more often combined with a *quantitative* analysis [3], by including *stochastic behaviour*, such as the fact that a message in a wireless communication scenario may get lost with probability p , and *real-time aspects*, e.g. the minimum and maximum times needed to transmit such a message, in models. This allows *quantitative* properties that relate to

* This work has been supported by the by the EU FP7 under grant number ICT-214755 (Quasimodo), by the German Research Council (DFG) as part of the Transregional Collaborative Research Center “Automatic Verification and Analysis of Complex Systems” (SFB/TR 14 AVACS) and by the DFG/NWO Bilateral Research Programme ROCKS.

Table 1. Submodels of STA [7]

Model	Probability distributions	Time	Nondeterminism
STA	arbitrary	arbitrary	yes
GSMP	arbitrary	arbitrary	no
PTA	finite-support	integer bounds	yes
TA	none	integer bounds	yes
PA/MDP	finite-support	no	yes
LTS	none	no	yes
MA/IMC	finite-support + exponential	exponential delays	yes
CTMC	finite-support + exponential	exponential delays	no
DTMC	finite-support	no	no

these additional features to be evaluated. In the communication scenario, examples could be “what is the worst-case probability of successful transmission of a message” (*probabilistic reachability*), “what is the expected time until the first message is received” (*expected-time reachability*), and even combinations, such as “what is the minimum probability of success within t time units” (probabilistic *time-bounded reachability*).

Such quantitative properties, in particular when time is involved, also allow the evaluation of a system’s *performance* in addition to *correctness*; on the other hand, a model involving probabilities and time also makes it possible to include quantitative requirements in correctness properties: “The protocol must ensure that the probability of success within 5 time units is larger than 95%.” Classic modal logics such as LTL or CTL have been adapted to these scenarios, resulting in logics such as PTCTL [20], which can be used to formally specify properties like the one given above.

In this paper, we review the use of stochastic timed automata (STA), the semantic foundation of the modelling language MODEST [7], as the modelling formalism of choice for systems combining stochastic and real-time aspects. STA are labelled transition systems enriched with clock variables to capture real-time aspects and probability distributions to cover stochastic behaviour. Many commonly used classes of automata can be seen as subclasses of STA, as summarized in Table 1 (where GSMP are generalised semi-Markov processes [12], (P)TA are (probabilistic) timed automata [1,20], PA/MDP are probabilistic automata [27] or Markov decision processes [26], LTS are labelled transition systems [4], MA/IMC are Markov automata [11] or interactive Markov chains [17], CTMC (DTMC) are continuous-time (discrete-time) Markov chains [4], and “finite-support” denotes probability distributions with finite support, such as the Bernoulli distribution or the discrete uniform distribution, “exponential” denotes the exponential distribution, while “exponential delays” denotes a model in which the times between event occurrences are or can be exponentially distributed).

In order to evaluate properties on a model, a variety of verification approaches are available, ranging from manual mathematical proofs or calculations over

semi-automated approaches like theorem proving to fully automatic techniques. Model-checking and simulation are the focus of this paper: They fall into the category of fully automatic techniques, which is a significant advantage from a usability perspective, and have already been applied successfully to a variety of the submodels of STA; yet, they are fundamentally different approaches:

Given a set of properties, an analysis with model-checking involves an exploration of the complete (reachable) state-space of the model. In the case of probabilistic models, to subsequently compute probabilities and obtain values for the properties, numerical techniques such as solving systems of linear equations or value iteration are necessary. In the simulation approach, on the other hand, a large number of concrete, finite paths of the model is explored, using random-number generators to resolve probabilistic choices. Only one state of the model is in memory at any time, and statistical techniques are used on the set of results for the individual runs to obtain mean values and confidence intervals.

The choice between model-checking and simulation is thus mainly a trade-off between memory consumption and time: Model-checkers generally need to represent the whole state-space in memory, but give accurate and precise (or at least safe) answers, while simulation needs constant memory for every model, even for infinite ones, but both the accuracy and the precision of the results depend directly (and exponentially) on the number of paths explored.

Unfortunately, both approaches are faced with problems when it comes to the analysis of STA: For model-checking, there is currently no technique available to deal with all the features of STA in one model, although large submodels can be model-checked (in particular PTA [10,16,18,19,21] and IMC [31]). In simulation, the problem is that *concrete* paths of the model are explored, and while the faithful random resolution of probabilistic choices works well with the final statistical evaluation of the collected results, there is no way to resolve nondeterministic choices without introducing additional assumptions. As it is, simulation can therefore only be used for deterministic models (read: GSMP in Table 1). While this is an inherent limitation of the approach, this paper aims to show ways of mitigating the problem for a practically relevant class of models.

The intention of this paper is to provide a thorough but compact introduction to STA (Section 2), to present a model-checking approach for one large and useful subclass, namely PTA [16], including a novel way to deal with *deadlines*, an aspect where STA differ from e.g. TA (Section 3), and to highlight a way to perform simulation for a superclass of GSMP that includes nondeterminism [6]—without introducing hidden assumptions or affecting the results (Section 4).

2 A Model for Stochastic Timed Systems

Stochastic timed automata [7] are a formal model for systems that include non-deterministic choices, finite probabilistic branching, real-time behaviour and decisions based on continuous probability distributions. They can be seen as an extension of various well-studied automata models, for example as the extension of timed automata (TA, [1]) with finite and continuous probabilistic choices. In

this section, we give a compact introduction to the model of STA, its semantics, and operators to support compositional modelling. However, let us first establish some preliminary notation.

Probability Distributions. A discrete *probability distribution* over a countable set Q is a function $\mu : Q \rightarrow [0, 1]$ such that $\sum_{q \in Q} \mu(q) = 1$. Let $\text{Dist}(Q)$ denote the set of all probability distributions over Q . The support of a distribution μ , $\text{support}(\mu)$, is the largest set $Q' \subseteq Q$ such that $\mu(q) > 0$ for all $q \in Q'$. We call $\mu \in \text{Dist}(Q)$ finite if $\text{support}(\mu)$ is finite. If its support is a singleton set $\{q\}$, μ is the point or Dirac distribution for q , denoted $\mathcal{D}(q)$.

For continuous distributions, standard measure theory is needed; we refer the reader to the wealth of textbooks on this matter (e.g. [15]) for details, and merely note that for this paper, $\text{Prob}(\Omega)$ denotes the set of all probability measures on $\mathcal{B}(\Omega)$, the Borel σ -algebra on the sample space Ω .

Clocks and Valuations. Our models contain real-valued variables, some of which are *clock variables*, or clocks. Given a set of variables Var , where Ck denotes the subset of clock variables, a valuation is a function $Var \rightarrow \mathbb{R}$ that assigns a concrete value to all variables. We let $Val(Var)$ denote the set of all valuations for the variables in Var ; $\mathbf{0}$ is the valuation that assigns 0 to all variables, and if $v \in Val$ and $t \in \mathbb{R}^+$, $v + t$ is the valuation where all clock variables have been incremented by t and all other variables remain unchanged.

Clock variables keep track of time, which advances at a constant rate that is the same for all clocks. *Clock constraints* are expressions of the form

$$CC ::= true \mid false \mid CC \wedge CC \mid CC \vee CC \mid c \sim x \mid c_1 \sim c_2 \mid x_1 \sim x_2,$$

where $\sim \in \{>, \geq, <, \leq, =, \neq\}$, $c, c_1, c_2 \in Ck$ and $x, x_1, x_2 \in \mathbb{R} \cup Var \setminus Ck$. For $e \in CC$ and $v \in Val$, we write $v(e)$ to denote the (Boolean) value of e evaluated in v . As usual, clocks can only be assigned to zero, *resetting* the clock.

2.1 Stochastic Timed Automata

We can now define the syntax and semantics of STA:

Syntax. A stochastic timed automaton is a tuple $(Loc, l_0, Act, Var, \rightarrow)$ where

- Loc is a set of *locations*,
- $l_0 \in Loc$ is the initial location.
- $Act = PAct \uplus IAct$ is a set of *actions*, partitioned into *patient* and *impatient* actions (see Section 2.2 for what this distinction is used for), where $\tau \in Act$ is the distinguished *silent action*,
- Var is a set of (real-valued) variables (with a subset $Ck \subseteq Var$ of *clock variables*), and
- $\rightarrow \subseteq Loc \times Act \times CC \times CC \times \text{Dist}(Asgn \times Loc)$ is the *edge relation*.

An edge $(l, a, g, d, \mu) \in \rightarrow$ consists of a *source location* l , an *action label* a , a *guard* g that determines when the edge is enabled (i.e. when it can be taken), a *deadline* d that imposes a condition on the passage of time (time cannot pass when a deadline in the current location is satisfied), and a *target probability distribution* μ over assignments and target locations.

Assignments are functions $Var \rightarrow Sxp$, where Sxp is the set of *sampling expressions*, which sample a value for a distinguished random variable $\xi \notin Var$ according to some distribution F . For example, if $\lambda \in Var$, $Exponential(\lambda + 3)$ represents a value sampled from the exponential distribution with rate $\lambda + 3$, while the sampling expression λ represents the value of λ , i.e. $\mathcal{D}(\lambda)$. (For a more formal definition, see [7].) For clarity, when writing assignments as sets of pairs in $Var \times Sxp$, we will leave out those pairs that leave a variable unchanged.

Two aspects of this model are worth noting explicitly: First, the real-time aspect of STA is based on timed automata with deadlines (TAD, [8]), where the passage of time is not constrained by *location invariants* as in classical TA, but deadlines (or *urgency constraints*) on the edges. Deadlines make it easier to avoid timelocks, and they are much more concise than invariants in certain scenarios when a model is specified as the parallel composition of several components; we will explore this relationship in more detail in Section 3.2.

Second, probabilistic choices appear at two points in STA: The target of an edge is a (discrete) probability distribution over assignments and locations, while the assignments themselves can contain sampling expressions, which may make use of (arbitrary) probability distributions. Continuous probabilistic choices are thus represented symbolically at the level of STA and will only become explicit in their semantics. The symbolic treatment of real-valued variables and continuous distributions allows STA to remain a *finite* model for “infinite” systems as long as the sets Loc , Act , Var and \rightarrow are finite.

Example 1. The STA in Figure 1 models a communication channel in which a message, after having been received at one end (impatient action `rcv` with guard *true* and deadline *false*), is lost with probability P_{loss} or transmitted to the other end with probability $1 - P_{loss}$, where it arrives (impatient action `snd`) after some (nondeterministically chosen) delay in the range $[TD_MIN, TD_MAX]$. Note how the deadline $c \geq TD_MAX$ of the edge labelled `snd` makes it impossible to stay in the upper-right state for more than TD_MAX time units.

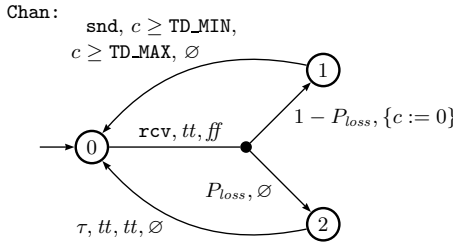


Fig. 1. STA model of a lossy communication channel

Semantics. Akin to TA, which have a semantics in terms of timed transition systems (TTS) with uncountably large state-spaces and sets of transition, the model of STA has an uncountable semantics with explicit continuous probability distributions in terms of timed probabilistic transition systems (TPTS):

A TPTS is a quadruple $(S, s_0, \Sigma, \hookrightarrow)$ where S is a nonempty (and usually uncountable) set of states, $s_0 \in S$ is the initial state, $\Sigma = Act \uplus \mathbb{R}^+$ is a set of transition labels, partitioned into actions in Act and delays in \mathbb{R}^+ , and $\hookrightarrow \subseteq S \times \Sigma \times \text{Prob}(S)$ is the stochastic transition relation. We write $s \xrightarrow{a} \mathbf{P}$ for $(s, a, \mathbf{P}) \in \hookrightarrow$. For every transition $(s, x, \mu) \in \hookrightarrow$ where $x \in \mathbb{R}^+$, we require that

- $\mu = \mathcal{D}(s')$,
- $(s, x, \mu') \in \hookrightarrow \Rightarrow \mu = \mu'$ (*time determinism*), and
- $(s, x+x', \mathcal{D}(s')) \in \hookrightarrow \Leftrightarrow (s, x, \mathcal{D}(s'')) \in \hookrightarrow \wedge (s'', x', \mathcal{D}(s')) \in \hookrightarrow$ (*additivity*).

The semantics of a STA $A = (Loc, l_0, Act, Var, \rightarrow)$ is the TPTS $\llbracket A \rrbracket = (Loc \times Val(Var), (l_0, \mathbf{0}), Act \uplus \mathbb{R}^+, \hookrightarrow_A)$ where \hookrightarrow_A is the smallest relation satisfying the following two inference rules:

$$\frac{l \xrightarrow{a,g,d} \mu \quad v(g) \text{ holds}}{(l, v) \xrightarrow{a}_A \lambda B. \sum_{A \in \text{Asgn}} \sum_{l' \in Loc} \mu((A, l')) \cdot \mathbf{A}_A^v(B)}$$

where \mathbf{A}_A^v , informally, samples the random variables in the assignment A and executes it (again, for a formal definition, see [7]), and

$$\frac{\forall t' < t: (v + t') (\neg \bigvee_{(l,a,g,d,\mu) \in \hookrightarrow} d) \text{ holds}}{(l, v) \xrightarrow{t} \mathcal{D}((l, v + t))}$$

The premise of the second rule is called the *time progress condition* for STA; it ensures that time can pass iff no deadline from the current location evaluates to *true*.

Discrete Variables. It is common to extend transition systems with variables that take values in some discrete domain D (see e.g. [4] Chapter 2] for the general recipe), and this can be lifted to STA. This allows the edges' target probability distributions to depend on these discrete variables; they can be removed by unrolling them into the locations just like the continuous variables are unrolled into the states of the underlying TPTS (see Figure 4 for an example).

2.2 Compositional Modelling

It is often useful to describe complex systems as the parallel composition of several independently specified interacting components. To enable compositional modelling with STA, we introduce process-algebraic expressions using a *parallel composition* operator that allows two STA to run in parallel, synchronising on a certain set of actions, and a *relabelling* operator that allows actions to be renamed for more flexible control of synchronisation. Parallel composition and

relabelling also exist in MODEST, and the operators we define here are at least as expressive as those in MODEST. We use the term *network of STA* to refer to such an expression as well as its semantics in terms of STA, while we use $[e]$ to specifically refer to its semantics. For brevity, we use $\llbracket e \rrbracket$ for $\llbracket [e] \rrbracket$.

Relabelling. Given a STA $A = (Loc, l_0, Act, Var, \rightarrow)$ and a *relabelling function* $f: Act \rightarrow Act$ with $f(\tau) = \tau$, the relabelling $f(A)$ is defined as $f(A) = (Loc, l_0, Act, Var, \rightarrow_f)$ where $(l, f(a), g, d, \mu) \in \rightarrow_f \Leftrightarrow (l, a, g, d, \mu) \in \rightarrow$.

Parallel Composition. Given a synchronisation alphabet $B \not\supseteq \{\tau\}$ and a pair of STA (A_1, A_2) with $A_i = (Loc_i, l_{0_i}, Act_i, Var_i, \rightarrow_i)$ with compatible signatures, i.e. $IAct_1 \cap PAct_2 = \emptyset$ and vice-versa, we let $Act = Act_1 \cup Act_2$ (and similarly for $IAct$ and $PAct$) and define the parallel composition $A_1 \parallel_B A_2$ as

$$A_1 \parallel_B A_2 = (Loc_1 \times Loc_2, (l_{0_1}, l_{0_2}), Act, Var_1 \cup Var_2, \rightarrow)$$

with $((l_1, l_2), a, g, d, \mu) \in \rightarrow$ if and only if

$$\begin{aligned} & a \notin B \wedge \exists (l_1, a, g, d, \mu_1) \in \rightarrow_1: \mu = \mu_1 \cdot \mathcal{D}((\emptyset, l_2)) \\ \text{or } & a \notin B \wedge \exists (l_2, a, g, d, \mu_2) \in \rightarrow_2: \mu = \mu_2 \cdot \mathcal{D}((\emptyset, l_1)) \\ \text{or } & a \in B \wedge \exists (l_i, a, g_i, d_i, \mu_i) \in \rightarrow_i \text{ for } i = 1, 2: \\ & g = g_1 \wedge g_2, d = d_1 \otimes d_2 \text{ and } \mu = \mu_1 \cdot \mu_2 \end{aligned}$$

where \cdot is defined as $(\mu_1 \cdot \mu_2)((A_1 \uplus A_2, (l_1, l_2))) = \mu_1((A_1, l_1)) \cdot \mu_2((A_2, l_2))$ and \otimes is \wedge for $a \in PAct$ and \vee for $a \in IAct$.

We may also allow global variables, which can be read and assigned by all STA contained in a network. In that case, synchronisation may lead to conflicting assignments to global variables; we consider this a modelling error and only consider models where such assignments do not occur.

3 Model-Checking

Model-checking currently appears feasible only for submodels of STA. As mentioned in the introduction, the two “largest” such submodels are PTA and IMC. We focus on model-checking for PTA, which combine discrete probabilistic choices, nondeterminism and real-time behaviour. The limitation of PTA compared to STA is that only probability distributions with finite support are allowed. While PTA are usually based on timed automata with location invariants, we show how to deal with deadlines as in STA in the second part of this section. Let us first illustrate the PTA model by introducing our running example for the remainder of this paper:

Example 2. A simple protocol for reliable data transfer over unreliable channels is the alternating bit protocol. We model the sender and receiver components of this protocol and the channel as STA which conform to the PTA subset, shown in Figures [1](#) and [2](#). Data is handled in an abstract manner, and only Boolean

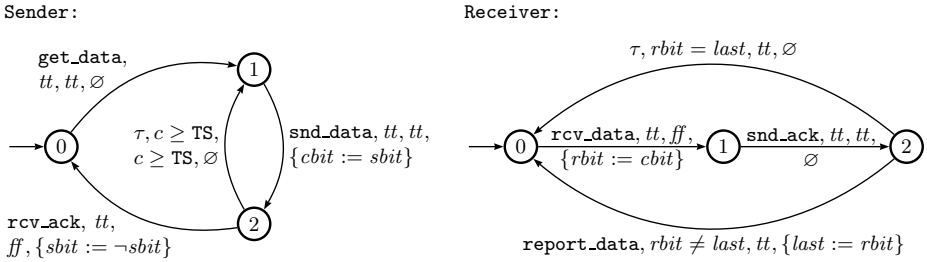


Fig. 2. A model of a sender and receiver using the alternating bit protocol

variables to keep track of the alternating bits are used. *sbit*, *rbit* and *last* are local variables; *cbit* is a global variable for the bit that is in transmission.

A complete model for this scenario is the network of STA given by

$$\text{Sender} \parallel_{\{\text{snd_data}, \text{rcv_ack}\}} (f_s(\text{Chan}) \parallel_{\emptyset} f_r(\text{Chan})) \parallel_{\{\text{rcv_data}, \text{snd_ack}\}} \text{Receiver}$$

with relabelling functions $f_s = \{ \text{rcv} \mapsto \text{snd_data}, \text{snd} \mapsto \text{rcv_data} \}$ and $f_r = \{ \text{rcv} \mapsto \text{snd_ack}, \text{snd} \mapsto \text{rcv_ack} \}$.

Because the communication channel we use does not reorder messages, it is not necessary to include a bit in acknowledgments. The sender’s environment will, from time to time (and only when the sender is ready), use the patient `get_data` action to indicate that a new message should be transmitted, while the receiver will report the arrival of fresh data via the `report_data` action.

3.1 A Modest Approach

The model-checking problem for PTA is well-understood and several algorithms exist, based on forwards [10,20] or backwards [21] reachability, digital clocks [19] or stochastic games [18]. We have developed a tool called `mcpta` to model-check PTA specified as STA in MODEST [16], which, by translating MODEST models (or, equivalently: networks of STA) into the input language of the PRISM probabilistic model-checker [24], allows to use the two most prominent approaches—stochastic games and digital clocks. In addition to “static” networks given by process-algebraic expressions as defined in this paper, a major contribution of `mcpta` is that it also works with *dynamic parallelism* (where existing components may spawn new parallel processes) in a compositional way.

Example 3. Using `mcpta`, we can, amongst others, verify the following properties on our running example as presented above:

$P_{\min/\max}$ The maximum/minimum (over all possible resolutions of nondeterministic choices) probability that n messages are reported by the receiver (via `report_data`). (1.0, 1.0)

$D_{\min/\max}$ The maximum/minimum probability that n messages are reported by the receiver within t time units. (0.996967, 0.999997)

$E_{\min/\max}$ The maximum/minimum expected time until n messages are reported by the receiver. (3.235448, 18.235439)

The model-checking results for $TD_MIN = 0$, $TD_MAX = 1$, $TS = 4$, $P_{loss} = 0.05$, $n = 8$, $t = 32$ are noted in parentheses; performance numbers can be found in Section 4.3. By appropriately adding some Boolean flags, we can also determine that, for example, the protocol does not work correctly when $TS \leq 2 \cdot TD_MAX$.

By using PRISM for the analysis of STA/PTA, however, we are using a tool that uses *location invariants* to represent restrictions on the passage of time instead of deadlines, which is why we investigated the relationship between deadlines and invariants:

3.2 Deadlines vs. Invariants

In standard (P)TA, the passage of time is constrained by location invariants, which are clock constraints associated to locations that allow time to pass as long as they are satisfied. However, invariants may easily lead to undesired timelocks, in particular in compositional models, and they cannot be used to represent certain forms of synchronisation [14].

In timed automata with deadlines [8], expressions associated to the edges control the passage of time, which makes it possible to avoid these problems and, for example, to build models that are timelock-free by construction. Several different ways to compose the time constraints in a compositional model are possible; of particular note is that deadlines allow *as-soon-as-possible* (ASAP) synchronisation of edges, which is useful when two edges in different components need to synchronise because their actions are part of the synchronisation alphabet, but have different guards that enable them after different points in time. ASAP synchronisation allows time to progress until both edges' guards are enabled, i.e. one edge waits for the other, without introducing timelocks.

Example 4. The **Sender** component of Example 2 is handed messages to transmit via the patient `get_data` action from some upper layer in the communication stack. A simple model for this upper layer is the STA A with one location l , one clock variable c and one edge (`get_data, c ≥ 5, c ≥ 5, D((c := 0), l)`). In **Sender** $\parallel_{\{\text{get_data}\}} A$, the edge labelled `get_data` is executed ASAP: A imposes a delay of 5 time units between occurrences of `get_data`, while the **Sender** may be busy with a previous transmission when the edge becomes enabled in A . Because $c \geq 5$ is a deadline, the result is that the edge is executed as soon as both 5 time units have elapsed since the previous execution *and* the sender is ready. If we instead associated the invariant $c \leq 5$ with l , A could stop the progress of time when the sender is still busy, thus potentially creating a timelock.

STA as defined in Section 2.1 exclusively use deadlines to control the passage of time. If we wanted to use invariants instead, we would modify the STA model as follows: The deadline component of the edge relation is dropped, and instead, an *invariant function* $i: Loc \rightarrow CC$ is added to the tuple defining an STA. In

Table 2. Converting between deadlines and invariants

deadline	$c > x$	$c \geq x$	$c < x$	$c \leq x$	$c_1 \sim c_2$	$c = x$	$c \neq x$	$c \leq x \wedge c \geq x$
invariant	$c \leq x$	$c \leq x$	$c \geq x$	$c > x$	$\neg(c_1 \sim c_2)$	–	$c = x$	\perp

the semantics, timed transitions are controlled by the following time progress condition (TPC) for invariants: $t > 0$ time units can pass in state (l, v) if

$$\forall t' \leq t: (v + t')(i(l)) \text{ holds.}$$

Contrast this to the TPC for STA with deadlines, which requires that

$$\forall t' < t: (v + t')(\neg \bigvee_{(l,a,g,d,\mu) \in \rightarrow} d) \text{ holds.}$$

A fine point here is the use of $<$ instead of \leq in the first quantification, which has some interesting consequences, for example that the deadlines $c > x$ and $c \geq x$ are equivalent, but also concerning the expressivity of deadlines.

In automata with invariants, an additional restriction that is commonly used is that edges cannot be followed if that would violate the invariant of the target location. If that restriction is (not) in effect, we speak of *strong (weak)* invariants. Strong invariants are semantically problematic in probabilistic models, since the target invariant may be violated for a strict subset of the support of the distribution sampled in an assignment. If strong invariants are desired, one solution is to require well-formed automata (e.g. well-formed PTA [21]).

A significant advantage of deadlines is the flexible choice of different operators for \otimes in parallel composition (Section 2.2)—for the invariant of a product location (l_1, l_2) , the only sensible choice is $i(l_1) \wedge i(l_2)$.

Expressivity. Due to the use of $<$ in the TPC for deadlines mentioned above, certain deadlines cannot be expressed as invariants and vice-versa. Table 2 summarises the differences: We already saw that the deadlines $c > x$ and $c \geq x$ are equivalent, and, given a location l with a single outgoing edge with such a deadline, it can be equivalently represented by $i(l) = c \leq x$. However, the invariant $c < x$ cannot be represented by any deadline, because deadlines cannot prevent the time point $c = x$ from being reached from below due to the use of $<$. For $c < x$, $c \leq x$ and all instances of $c_1 \sim c_2$, the translation between deadline and invariant is straightforward based on the differences in the TPCs.

$c = x$ and $c \neq x$ are interesting cases: The deadline $c \neq x$ is equivalent to the invariant $c = x$, but both the deadline $c = x$ and the invariant $c \neq x$ do not have a corresponding equivalent. The reason for the latter is the same as for the invariant $c < x$, while the deadline $c = x$ is a curious case: Starting at a point in time where $c \leq x$, the deadline $c = x$ allows time to progress just until $c = x$ is reached, but no further. Starting at $c > x$, time progress is not constrained. This behaviour cannot be represented by an invariant: In order to prevent time from passing when $c = x$, there must be an $\epsilon > 0$ such that the interval $]x, x + \epsilon[$

is not included in the invariant, but this would prevent time from passing when already $c > x$, e.g. when $c = x + \frac{1}{2}\epsilon$.

The first six columns of Table 2 define a conversion function $Conv: CC \rightarrow CC$ to convert atomic clock constraints from deadlines to equivalent invariants (*true*, *false* and $x_1 \sim x_2$ are simply negated). This function can be lifted to composite clock constraints by applying it to the atomic subexpressions and flipping the Boolean operators ($\wedge \mapsto \vee, \vee \mapsto \wedge$). However, this lifting fails if the deadline $c = x$ is encoded in an “obfuscated” way as shown in the table’s last column: $Conv(c \leq x \wedge c \geq x) = c > x \vee c \leq x = true$, but the deadline $c \leq x \wedge c \geq x$ cannot be represented as an invariant.

From Invariants to Deadlines. A STA with invariants that can all be expressed as deadlines can be transformed into an automaton with deadlines by simply adding a permanently disabled loop $(l, \tau, false, \neg i(l), \mathcal{D}(\emptyset, l))$ to every location l , making use of the fact that disabled guards do not influence the effect of deadlines, and setting the deadline of all other edges to *false*. If a strong invariant semantics is desired, the guards of all these edges must be set to $g \wedge i(l)$ to prevent any edge from being taken when the invariant is violated, where g is the edge’s guard in the original STA with invariants. This transformation is also sufficient for a network of automata when performed for each of the components. A more detailed explanation and a correctness proof can be found in [7, Section VI].

From Deadlines to Invariants. For a single STA whose deadlines are all (and, to avoid the compositionality problem with $Conv$ shown above, in combination) expressible as invariants, the transformation into a STA with weak invariants is straightforward and follows from the TPCs: Set $i(l) = \bigwedge_{(l,a,g,d,\mu) \in \rightarrow} Conv(d)$ and keep everything else as-is. Due to the flexible handling of deadlines in parallel composition with patient and impatient actions, however, transforming a network of STA into a network of STA with invariants is much more complicated.

Example 5. Figure 3 shows two STA with deadlines and their translation into invariants (next to the locations). If we just transform the component STA into STA with deadlines as described above, the resulting parallel composition is wrong, as shown on the rightmost automaton: The deadline $x > 3$ should not have an effect on location 0A, and if **a** is a patient action (i.e. $\otimes = \wedge$), the invariant of location 1A is also incorrect—it should be a disjunction, but invariants only support conjunction in parallel composition.

We thus need to take the context in terms of automata in parallel compositions and relabelings into account to transform deadlines for transitions labelled a into invariants that only apply when an edge labelled a is actually available, i.e. all automata that must synchronise on a can do so (to solve the problem of location 0A), and that use the correct operators to compose deadlines (to solve the problem of location 1A).

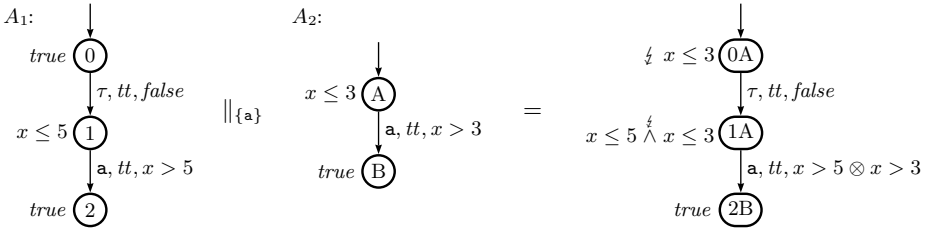


Fig. 3. Converting deadlines to invariants compositionally (failed attempt)

In order to achieve this without flattening a network of STA into a single STA (with invariants), we compute a global invariant $gi(\hat{e})$ for the process-algebraic expression \hat{e} describing the network, and add a single-location STA GI with invariant $gi(\hat{e})$ to the composition, while all other locations' invariants are set to *true*. We can compute this global invariant recursively from the subexpressions as follows:

$$\begin{aligned}
 gi(e) &= \bigwedge_{a \in Act_{[e]}} gi_a(e) \\
 gi_a(e_1 \parallel_B e_2) &= \begin{cases} En_a(e_1) \wedge En_a(e_2) \Rightarrow gi_a(e_1) \bar{\otimes} gi_a(e_2) & \text{if } a \in B \\ gi_a(e_1) \wedge gi_a(e_2) & \text{otherwise} \end{cases} \\
 gi_a(f(e)) &= \bigwedge_{b: f(b)=a} gi_b(e) \\
 gi_a(A) &= \bigwedge_{l \in Loc_A} \left(P_l^A \Rightarrow \bigwedge_{(l,a,g,d,\mu) \in \rightarrow_A} Conv(d) \right)
 \end{aligned}$$

where $P_l^A \Leftrightarrow A$ is in location l , $\bar{\otimes}$ is \vee if a is patient and \wedge otherwise, and $En_a(e)$ is a predicate that characterises the locations in which an edge labelled a is available in the automaton represented by e :

$$\begin{aligned}
 En_a(e_1 \parallel_B e_2) &= \begin{cases} En_a(e_1) \wedge En_a(e_2) & \text{if } a \in B \\ En_a(e_1) \vee En_a(e_2) & \text{otherwise} \end{cases} \\
 En_a(f(e)) &= \bigvee_{b: f(b)=a} En_b(e) \\
 En_a(A) &= P_a^A
 \end{aligned}$$

where P_a^A is *true* iff the STA A is in a location with an outgoing edge labelled a .

Lemma 1. For a process-algebraic expression e , $\llbracket e \rrbracket$ (with deadlines) and $\llbracket GI \parallel_{\emptyset} e \rrbracket$ (with invariants computed as described above) are isomorphic.

Proof. By induction over the structure of e .

Example 6. For the STA in Example 5, assuming that a is patient, the global invariant computed as described above is

$$P_a^{A_1} \wedge P_a^{A_2} \Rightarrow (P_1^{A_1} \Rightarrow x \leq 5) \vee (P_A^{A_2} \Rightarrow x \leq 3).$$

Since $P_a^{A_1} \Leftrightarrow P_1^{A_1}$ and $P_a^{A_2} \Leftrightarrow P_A^{A_2}$, it is equivalent to what we intuitively expect:

$$P_1^{A_1} \wedge P_A^{A_2} \Rightarrow x \leq 5 \vee x \leq 3$$

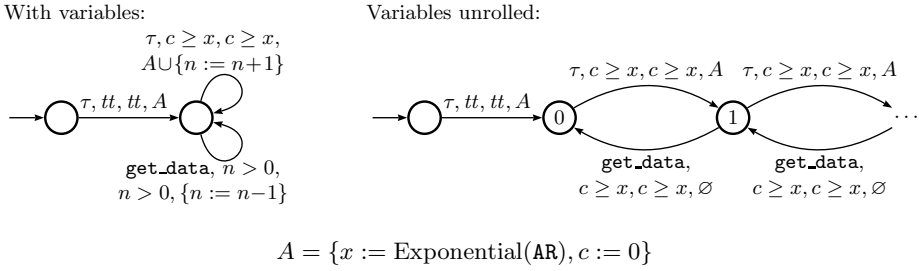


Fig. 4. STA model of a queue based on a Poisson process with arrival rate AR

4 Simulation

Using simulation, finite and infinite models with arbitrary probability distributions as well as advanced modelling features such as dynamic networks of automata or complex recursive functions can be analysed; on the other hand, a simulation analysis can only deliver approximate results, since it is not guaranteed to explore the whole state-space. However, this is where its most significant advantage comes from: Simulation needs only constant memory, to represent the current state during the exploration of a path through the model. Simulation is at the core of other analysis techniques like statistical model-checking [5,30,32].

Example 7. Instead of the simple upper layer introduced in Example 4, we can model it as queue that stores messages that should be sent, but have not yet been processed by the sender, and to which new messages arrive in intervals of exponentially distributed length. Figure 4 shows the STA model of such a queue with arrival rate AR. If we add this to Example 2, the result is a model with continuous probability distributions and an infinite number of locations in addition to the discrete probabilities, real-time aspects and nondeterminism already present previously. It is thus no longer in any of the true submodels of STA that we presented, and in particular, no longer amenable to model-checking; only a simulation analysis appears feasible.

Unfortunately, models with nondeterministic choices cannot be simulated—simulation relies on the model being a stochastic process. This requirement is often hidden by popular simulation tools, using various kinds of assumptions, which leads to problems of its own [9]. In the following, we will first highlight the problems of simulating nondeterminism and using hidden assumptions, and then present a solution that guarantees “surprise-free” simulation for nondeterministic models like our running example.

4.1 Resolving Nondeterminism

Let us now investigate the consequences of just resolving nondeterminism in some well-defined way and then simulating the resulting deterministic model:

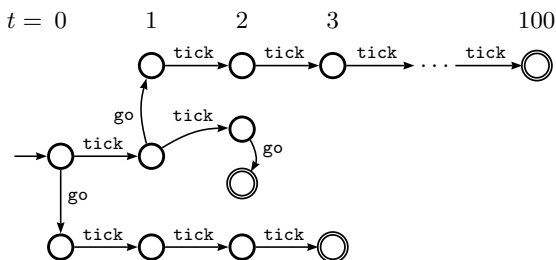


Fig. 5. An anomalous discrete-timed system

If we compute the probability of a set of paths for a particular way to resolve nondeterministic choices, we only obtain the probability for the subset of the system’s behaviour induced by that resolution of choices, which may or may not be useful, but which can also be very misleading:

After all, there are three typical uses for nondeterminism: First, in case of complete absence of knowledge about a certain choice—not even some probabilities are known—that choice can be modelled as a nondeterministic one. Second, in a refinement process where abstract models are progressively refined to more and more concrete implementations, a nondeterministic choice may leave open certain choices. Finally, nondeterminism can allow an unspecified environment to make certain choices in an open model. In the latter two cases, obtaining results for *some* environment or *some* implementation is not particularly useful; in fact, if the result happens to be very optimistic (e.g. by not considering some few adverse environments or unfortunate implementations), it may lead to unfounded conclusions that may jeopardise the safety of the actual system whose study the model was built for. Only in the first case does a uniformly random resolution of the choice make some sense, but there is still the risk of catastrophic behaviours being balanced by excellent ones, leading to an acceptable average result.

Example 8. The discrete-time LTS in Figure 5 contains two nondeterministic choices between action `go` and letting time pass. Let the property of interest be the expected time until a goal state (indicated by a double circle) is reached. The results we would obtain via model-checking would be a minimum (best-case) time of 2 ticks and a maximum (worst-case) time of 100 ticks. For a simulation analysis, the nondeterminism needs to be resolved. If we resolve it in a uniformly distributed random manner, the result would be around 27 ticks. Note that this is quite far from the actual worst-case behaviour, and in particular, by adding more “fast” or “slow” alternatives to the model, we can change the result arbitrarily. Even worse, a very small change to the model can make a quite a difference: If the `go`-labelled transition to the upper branch were available in the initial state instead of after one `tick`, the “uniform” result would have been 35 ticks.

There are more sophisticated strategies to resolve nondeterminism here; knowing that the `tick` action models the passage of time, we could try scheduling the transitions labelled `go` as soon or as late as possible (“ASAP” and “ALAP”).

Intuitively, we might expect to obtain the best-case behaviour using the ASAP and the worst-case behaviour using the ALAP scheduler. Unfortunately, the results run counter to this intuition—ASAP simulation yields an expected time of 3 ticks, while ALAP yields the best-case result of 2 ticks, and the worst case of 100 ticks is completely ignored. This is because the example exhibits a timing anomaly: It is best to wait as long as possible before scheduling `go` in order to obtain the minimum expected time. For this toy example, the anomaly can easily be seen by looking at the model, but similar effects may of course occur in complex models where they cannot easily be detected.

We therefore argue that the only safe way to simulate models containing actual nondeterministic choices is to not simulate them at all. In particular, using some resolution method under-the-hood in a simulation tool—without warning the user of the possible consequences—is dangerous.

Parallel Composition. Unfortunately, even if the user is aware that nondeterministic models are not suitable for simulation and is careful to make all the components of his or her network-of-STA model deterministic, nondeterminism may still be introduced by the parallel composition operation: If more than one component STA has an enabled, non-synchronising transition, the order of execution is unspecified—this is the usual interleaving semantics. Having said that, the reverse may also happen: By synchronising via shared actions or global variables, the behaviour of a network of nondeterministic STA may itself be deterministic. It is however very difficult to build a large model out of components such that the result is surely deterministic.

Example 9. The **Channel**, **Sender** and **Receiver** components of the model presented in Example 2 are deterministic, but the parallel composition is not: For example, if the receiver is in location 2 and the acknowledgment channel in location 2 (i.e. they have just synchronised on `snd_ack` and the channel has decided to lose the message), two edges are immediately available in the composition: The receiver can return to location 0 via `report_data` or τ , and the channel can return to location 0 via τ . The composition is therefore nondeterministic.

4.2 Partial-Order Methods for Simulation

From a user perspective, the previous conclusion that models containing actual nondeterministic choices must not be simulated is a highly unsatisfactory one; after all, many models are too expressive or simply too large to be handled with model-checking approaches, and simulation is the only feasible analysis technique. Fortunately, the word *actual* is key here: What if all nondeterministic choices present in the model do not actually affect the simulation results, i.e. all nondeterminism is instead *spurious*? In such a case, simulation is safe; the question is how such a situation can be identified without negating the benefits in terms of memory usage of the simulation approach.

Example 10. The nondeterminism highlighted in Example 9 does not affect the time until the first (or any) message is successfully transmitted—it does not matter whether the channel first completes the loss of the message or the receiver first returns to its initial location since the receiver now has to wait for TS time units in any case. For all possible resolutions of this nondeterministic choice, the value of the property remains unchanged; this nondeterminism is thus spurious. In fact, careful manual investigation of this (rather small) model will show that all nondeterminism except for the choice of transmission delay in $[TD_MIN, TD_MAX]$ is spurious for the properties we consider in this paper.

It turns out that such a spuriousness check can be performed automatically and on-the-fly during simulation. Our approach to this is inspired by partial order reduction [2,13,25,28], a technique used in model-checking to reduce a model's state-space by removing spurious nondeterministic choices, keeping only one representative for the relevant behaviour. As usually implemented in model-checking tools, partial order reduction is an overapproximation in the sense that it cannot identify all spurious choices as such, but those that it does identify are guaranteed to be spurious. In particular, only nondeterminism that results from interleaving due to parallel composition can be identified as spurious. The advantage over other (optimal) minimisation approaches, e.g. bisimulation minimisation, is that these partial order implementations do not need a representation of the entire state-space of the model; they can be incorporated into the state-space generation procedure to construct the smaller state-space on-the-fly with information from the compositional description of the model.

This makes it possible to adapt the existing partial order reduction technique for PA/MDP [2] to the simulation setting: Simulation proceeds as usual until a nondeterministic choice is encountered. Whenever that is the case, the partial order method is invoked to check which of the alternatives can safely be eliminated; if all but one can, simulation proceeds with that one. The algorithm of [2] needs a certain lookahead to, intuitively, determine whether all paths for the different choices lead to the same result (state) without conflicts. The amount of states explored depends on the model, but is usually significantly smaller than the entire state-space. Still, for simulation, which can also deal with infinite-state models, care must be taken to make sure that even in the worst case, the amount of states explored here is bounded. Introducing such an exploration depth bound, denoted k in the remainder of this paper, results in an additional level of overapproximation, since we have to consider a choice as truly nondeterministic and abort simulation when the bound is exceeded; yet, in the case studies we considered, small values of k (< 10) were sufficient to identify all nondeterminism as spurious.

The approach described above has been implemented in *modes*, our discrete-event simulator for MODEST. We refer the reader to [6] for the formal and technical details; in the remainder of this section, we instead briefly sketch how we extend the approach from PA/MDP to STA in *modes*, and we investigate its usefulness using our running example.

Table 3. Model-checking and simulation performance

model-checking with <code>mcpta</code>					model		simulation with <code>modes</code>				
states	time	memory	ϵ	TD	n	t	TD	runs	time	\varnothing_E	\varnothing_D
14 306	4 s	0.5 MB	10^{-6}	[0, 1]	8	32	[1, 1]	1100	4 s	10^{-2}	10^{-2}
55 130	13 s	2.2 MB	10^{-6}	[0, 1]	16	64	[1, 1]	2000	13 s	10^{-2}	–

Time and Continuous Distributions. The partial order reduction approach of [2] is specified for the model of PA (or, equivalently, MDP). There are, to our knowledge, no results on partial order reduction techniques that deal with real time or continuous probabilistic choices[4], which are an essential part of STA. However, we can still use the approach for PA by treating the additional features of STA in an orthogonal way:

The problem with *time* is that its passage introduces an implicit synchronisation over all component automata by incrementing the values of all clocks. `modes` acknowledges this fact by providing a separate choice of scheduler for time and treating resulting non-zero deterministic delay steps like visible transitions. Nondeterminism can thus be detected as spurious only if the interleaving happens in zero time. In order to correctly detect the spuriousness of nondeterminism in presence of assignments involving sampling expressions with *continuous probability distributions*, `modes` overapproximates by treating them like a nondeterministic assignment to some value from the distribution’s support.

4.3 Simulating the Communication Example

We have seen in Examples 9 and 10 that our running example contains nondeterminism, but if we set `TD_MIN = TD_MAX (= 1)`, it is all spurious. It can thus be simulated using the partial-order based method presented above, and we present the simulation results in the remainder of this section.

`mcpta`, `modes` and PRISM were run on a single-core Intel Pentium M 1.7 GHz system with 1 GB of RAM in all of the following experiments.

Compared to Model-Checking. We can use `modes` to simulate the running example without queue for the properties presented in Example 3 except for properties `P...` (they do not include a time bound, yet simulation explores only finite paths). A performance comparison for this toy example can be found in Table 3; we chose the number of simulation runs to make simulation and model-checking times approximately equal. ϵ is the relative error accepted by PRISM’s convergence check in value iteration, while \varnothing_E and \varnothing_D are the dimensions of the width of the 95% confidence interval computed by `modes` for the `D.../E...` properties, respectively. We cannot give representative values for memory

¹ All approaches for TA (see e.g. Minea [22] for one approach and an overview of related work) rely on modified semantics for TA, on models more restricted in the timed behaviour, or on severe restrictions of the kinds of properties preserved. We are not aware of any approaches for models with continuous distributions.

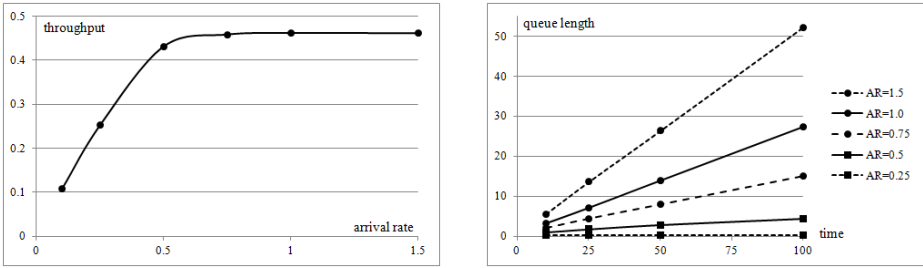


Fig. 6. Simulation results for the communication scenario

consumption for simulation due to the garbage-collected environment `modes` runs in, but it appears to be equal for both models.

With Poisson Arrivals. A more interesting case for simulation is the running example with queue introduced in Example 7 because model-checking (with currently available approaches) is not possible for this model. Due to the spuriousness of the nondeterminism, simulation still works, and also supports the analysis of new kinds of properties. This includes the average throughput (successful message transmissions per time) and the average queue length, both up to a certain time bound.

The results are plotted in Figure 6 (where throughput is measured for the first 100 time units). We see that the throughput of the system is limited to about 0.46 messages per time unit due to the communication delays and the lengths of the timeouts in case a message is lost; as expected, the queue length appears to grow without bound for arrival rates close to or larger than that value; slowly so for $AR = 0.5$, but already significantly for $AR = 0.75$.

Larger case studies and a more detailed performance evaluation of the partial order-based on-the-fly approach can be found in [6]. A more thorough comparison of model-checking and a simulation-based approach is presented in [29].

5 Conclusion

In this paper, we have presented stochastic timed automata (STA) as an overarching formalism to model and analyse stochastic timed systems. STA are very expressive, and using the modelling language MODEST or the process-algebraic constructs introduced in this paper also allows the convenient expression of compositional models, which makes models more readable and the modelling task easier to handle.

We have also highlighted two approaches for the analysis of STA: Model-checking for the subset of STA corresponding to probabilistic timed automata and simulation for networks of STA where the analysis results do not depend on any nondeterministic choices. For the former, we have shown how to handle deadlines in a setting that only allows location invariants, and for the latter,

we sketched a way to make simulation work in the presence of certain forms of nondeterminism. Both approaches are still restricted to, albeit large, submodels of STA, but have already proven to be useful. We also note that simulation is at the core of other analysis approaches such as statistical model-checking, which thus also benefit from these results.

The tools implementing these approaches, `mcpta` and `modes`, are available at www.modestchecker.net as part of the MODEST TOOLSET, which aims to provide a comprehensive, portable and easy-to-use environment for the analysis of MODEST models and thus STA.

Acknowledgments. The results presented in this paper are based on work by and discussions with Henrik Bohnenkamp, Pedro D’Argenio, Holger Hermanns and Joost-Pieter Katoen concerning MODEST and the model of STA [7], on research performed with Holger Hermanns regarding model-checking for PTA [16], and on work performed in collaboration with Jonathan Bogdoll, Luis María Ferrer Fioriti and Holger Hermanns on simulation for nondeterministic models [6].

References

1. Alur, R., Dill, D.L.: A theory of timed automata. *Theor. Comput. Sci.* 126(2), 183–235 (1994)
2. Baier, C., D’Argenio, P.R., Größer, M.: Partial order reduction for probabilistic branching time. *Electr. Notes Theor. Comput. Sci.* 153(2), 97–116 (2006)
3. Baier, C., Haverkort, B.R., Hermanns, H., Katoen, J.P.: Performance evaluation and model checking join forces. *Commun. ACM* 53(9), 76–85 (2010)
4. Baier, C., Katoen, J.P.: *Principles of Model Checking*. MIT Press, Cambridge (2008)
5. Basu, A., Bensalem, S., Bozga, M., Caillaud, B., Delahaye, B., Legay, A.: Statistical abstraction and model-checking of large heterogeneous systems. In: Hatcliff, J., Zucca, E. (eds.) *FMOODS/FORTE 2010*. LNCS, vol. 6117, pp. 32–46. Springer, Heidelberg (2010)
6. Bogdoll, J., Ferrer Fioriti, L.M., Hartmanns, A., Hermanns, H.: Partial order methods for statistical model checking and simulation. In: Bruni, R., Dingel, J. (eds.) *FMOODS/FORTE 2011*. LNCS, vol. 6722, pp. 59–74. Springer, Heidelberg (2011)
7. Bohnenkamp, H.C., D’Argenio, P.R., Hermanns, H., Katoen, J.P.: MoDeST: A compositional modeling formalism for hard and softly timed systems. *IEEE Transactions on Software Engineering* 32(10), 812–830 (2006)
8. Bornot, S., Sifakis, J.: An algebraic framework for urgency. *Inf. Comput.* 163(1), 172–202 (2000)
9. Cavin, D., Sasson, Y., Schiper, A.: On the accuracy of MANET simulators. In: *POMC*, pp. 38–43. ACM, New York (2002)
10. Daws, C., Kwiatkowska, M.Z., Norman, G.: Automatic verification of the IEEE 1394 root contention protocol with KRONOS and PRISM. *STTT* 5(2-3), 221–236 (2004)
11. Eisentraut, C., Hermanns, H., Zhang, L.: On probabilistic automata in continuous time. In: *LICS*, pp. 342–351. IEEE Computer Society, Los Alamitos (2010)
12. Glynn, P.W.: A GSMP formalism for discrete event systems. *Proceedings of the IEEE* 77, 14–23 (1989)

13. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem. LNCS, vol. 1032. Springer, Heidelberg (1996)
14. Gómez, R.: A compositional translation of timed automata with deadlines to UP-PAAL timed automata. In: Ouaknine, Vaandrager (eds.) [23], pp. 179–194
15. Grimmet, G., Stirzaker, D.: Probability and Random Processes, 3rd edn. Oxford University Press, Oxford (2001)
16. Hartmanns, A., Hermanns, H.: A Modest approach to checking probabilistic timed automata. In: QEST, pp. 187–196. IEEE Computer Society, Los Alamitos (2009)
17. Hermanns, H.: Interactive Markov Chains: The Quest for Quantified Quality. LNCS, vol. 2428. Springer, Heidelberg (2002)
18. Kwiatkowska, M.Z., Norman, G., Parker, D.: Stochastic games for verification of probabilistic timed automata. In: Ouaknine, Vaandrager (eds.) [23], pp. 212–227
19. Kwiatkowska, M.Z., Norman, G., Parker, D., Sproston, J.: Performance analysis of probabilistic timed automata using digital clocks. *Formal Methods in System Design* 29(1), 33–78 (2006)
20. Kwiatkowska, M.Z., Norman, G., Segala, R., Sproston, J.: Automatic verification of real-time systems with discrete probability distributions. *Theor. Comput. Sci.* 282(1), 101–150 (2002)
21. Kwiatkowska, M.Z., Norman, G., Sproston, J., Wang, F.: Symbolic model checking for probabilistic timed automata. *Inf. Comput.* 205(7), 1027–1077 (2007)
22. Minea, M.: Partial order reduction for model checking of timed automata. In: Baeten, J.C.M., Mauw, S. (eds.) CONCUR 1999. LNCS, vol. 1664, pp. 431–446. Springer, Heidelberg (1999)
23. Ouaknine, J., Vaandrager, F.W. (eds.): FORMATS 2009. LNCS, vol. 5813. Springer, Heidelberg (2009)
24. Parker, D.: Implementation of Symbolic Model Checking for Probabilistic Systems. Ph.D. thesis, University of Birmingham (2002)
25. Peled, D.: Combining partial order reductions with on-the-fly model-checking. In: Dill, D.L. (ed.) CAV 1994. LNCS, vol. 818, pp. 377–390. Springer, Heidelberg (1994)
26. Puterman, M.L.: Markov Decision Processes: Discrete Stochastic Dynamic Programming. Wiley Series in Probability and Mathematical Statistics: Applied Probability and Statistics. John Wiley & Sons Inc., New York (1994)
27. Segala, R.: Modeling and Verification of Randomized Distributed Real-Time Systems. Ph.D. thesis, MIT, Cambridge, MA, USA (1995)
28. Valmari, A.: A stubborn attack on state explosion. In: Clarke, E.M., Kurshan, R.P. (eds.) CAV 1990. LNCS, vol. 531, pp. 156–165. Springer, Heidelberg (1991)
29. Younes, H.L.S., Kwiatkowska, M.Z., Norman, G., Parker, D.: Numerical vs. Statistical probabilistic model checking: An empirical study. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 46–60. Springer, Heidelberg (2004)
30. Younes, H.L.S., Simmons, R.G.: Probabilistic verification of discrete event systems using acceptance sampling. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 223–235. Springer, Heidelberg (2002)
31. Zhang, L., Neuhäuffer, M.R.: Model checking interactive markov chains. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 53–68. Springer, Heidelberg (2010)
32. Zuliani, P., Platzer, A., Clarke, E.M.: Bayesian statistical model checking with application to simulink/stateflow verification. In: Johansson, K.H., Yi, W. (eds.) HSCC, pp. 243–252. ACM, New York (2010)

Author Index

- Anis Mekki, Mohamed 23
- Baresi, Luciano 267
- Beckert, Bernhard 61
- Beohar, Harsh 316
- Borner, Thorsten 61
- Bravetti, Mario 165
- Broch Johnsen, Einar 142
- Bubel, Richard 80
- Butler, Michael 122, 251
- Chevalier, Yannick 23
- Clarke, Dave 204
- Cuijpers, Pieter 316
- David, Alexandre 336, 352
- Donaldson, Alastair F. 297
- Eriksson, Henrik 287
- Fontejn, Jasper 316
- Groza, Bogdan 45
- Grunnet, Jacob Deleuran 336
- Gurov, Dilian 184
- Hähnle, Reiner 80, 142
- Hartmanns, Arnd 372
- He, Nannan 297
- Hvid Hansen, Helle 225
- Jessen, Jan Jakob 336
- Ji, Ran 80
- Ketema, Jeroen 225
- Klebanov, Vladimir 61
- Kroening, Daniel 297
- Lanese, Ivan 165
- Larsen, Kim Guldstrand 336, 352
- Li, Shuhao 352
- Lienhardt, Michaël 165
- Luttik, Bas 225
- Marchi dos Santos, Osmar 225
- Mikucionis, Marius 352
- Minea, Marius 45
- Mödersheim, Sebastian 1
- Morzenti, Angelo 267
- Motta, Alfredo 267
- Mousavi, MohammadReza 225
- Muschevici, Radu 204
- Nadales Agut, Damian 316
- Nielsen, Brian 352
- Petre, Luigia 102
- Poetzsch-Heffter, Arnd 165
- Proença, José 204
- Rasmussen, Jacob Illum 336
- Rossi, Matteo 267
- Rümmer, Philipp 297
- Rusinowitch, Michaël 23
- Sangiorgi, Davide 165
- Savicks, Vitaly 251
- Schaefer, Ina 184, 204
- Schäfer, Jan 142, 165
- Schlatte, Rudolf 142, 204
- Sere, Kaisa 102
- Silva, Renato 122
- Snook, Colin 251
- Soleimanifard, Siavash 184
- Steffen, Martin 142
- Svenningsson, Rickard 287
- Törngren, Martin 287
- Tsiopoulos, Leonidas 102
- van Beek, Bert 316
- van de Pol, Jaco 225
- Vinter, Jonny 287
- von Oheimb, David 1
- Welsch, Yannick 165
- Zavattaro, Gianluigi 165