Sixth International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2007)

ESEC/FSE 2007
6th Joint Meeting of the European Conference on Software Engineering and the ACM SIGSOFT Symposium on the Foundations of Software Engineering
Dubrovnik, Croatia
September 3-4, 2007

SAVCBS 2007 PROCEEDINGS

Specification and Verification of Component-Based Systems

http://www.eecs.ucf.edu/SAVCBS/

September 3-4, 2007
Dubrovnik, Croatia

Workshop at ESEC/FSE 2007
6th Joint Meeting of the
European Conference on Software Engineering and the
ACM SIGSOFT Symposium on the
Foundations of Software Engineering

Sixth International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2007), September 3-4, 2007, Dubrovnik, Croatia.

SAVCBS 2007
TABLE OF CONTENTS

SAVCBS 2007
ORGANIZING COMMITTEE

Mike Barnett (Microsoft Research, USA)
Mike Barnett is a Research Software Design Engineer in the Foundations of Software Engineering group at Microsoft Research. His research interests include software specification and verification, especially the interplay of static and dynamic verification. He received his Ph.D. in computer science from the University of Texas at Austin in 1992.

Dimitra Giannakopoulou (RIACS/NASA Ames Research Center, USA)
Dimitra Giannakopoulou is a RIACS research scientist at the NASA Ames Research Center. Her research focuses on scalable specification and verification techniques for NASA systems. In particular, she is interested in incremental and compositional model checking based on software components and architectures. She received her Ph.D. in 1999 from the Imperial College, University of London.

Gary T. Leavens (School of EECS, University of Central Florida, USA)
Gary T. Leavens is a professor in the School of Electrical Engineering and Computer Science at the University of Central Florida. He moved to Orlando in Fall 2007. Previously he was a professor of Computer Science at Iowa State University. His research interests include programming and specification language design and semantics, program verification, and formal methods, with an emphasis on the object-oriented and aspect-oriented paradigms. He received his Ph.D. from MIT in 1989.

Natasha Sharygina (CMU and SEI, USA; Lugano, Switzerland)
Natasha Sharygina is a senior researcher at the Carnegie Mellon Software Engineering Institute and an adjunct assistant professor in the School of Computer Science at Carnegie Mellon University, and an assistant professor at the University of Lugano. Her research interests are in program verification, formal methods in system design and analysis, systems engineering, semantics of programming languages and logics, and automated tools for reasoning about computer systems. She received her Ph.D. from The University of Texas at Austin in 2002.

SAVCBS 2007
PROGRAM COMMITTEE

Arnd Poetzsch-Heffter (Department of CS, Univ. of Kaiserslautern)
Arnd Poetzsch-Heffter chaired the program committee for SAVCBS 2007. He is a professor in the Department of Computer Science at the University of Kaiserslautern, Germany. His research interests are in component-oriented programming, program verification and generative programming. He received his Ph.D. and Habilitation Degree in Computer Science from the Technische Universität München in 1991 and 1997.

Workshop Program Committee:
Jonathan Aldrich (Carnegie Mellon University)
Michael Barnett (Microsoft Research)
Marcello M. Bonsangue (LIACS – Leiden University)
Paulo Borba (Federal University of Pernambuco)
Kathi Fisler (WPI)
Cormac Flanagan (University of California, Santa Cruz)
Marieke Huisman (INRIA Sophia Antipolis)
Joost-Pieter Katoen (RWTH Aachen)
Gary T. Leavens (Iowa State University)
Peter Müller (ETH Zürich)
David Naumann (Stevens Institute of Technology)
Matthew Parkinson (University of Cambridge)
Arnd Poetzsch-Heffter (University of Kaiserslautern), PC Chair
Ralf Reussner (Universität Karlsruhe)
Natasha Sharygina (Lugano and Carnegie Mellon)
Kurt C. Wallnau (Software Engineering Institute)
Tao Xie (North Carolina State)

SAVCBS 2007
WORKSHOP INTRODUCTION

This workshop is concerned with how formal (i.e., mathematical) techniques can be or should be used to establish a suitable foundation for the specification and verification of component-based systems. Component-based systems are a growing concern for the software engineering community. Specification and reasoning techniques are urgently needed to permit composition of systems from components. Component-based specification and verification is also vital for scaling advanced verification techniques such as extended static analysis and model checking to the size of real systems. The workshop will consider formalization of both functional and non-functional behavior, such as performance or reliability.

This workshop brings together researchers and practitioners in the areas of component-based software and formal methods to address the open problems in modular specification and verification of systems composed from components. We are interested in bridging the gap between principles and practice. The intent of bringing participants together at the workshop is to help form a community-oriented understanding of the relevant research problems and help steer formal methods research in a direction that will address the problems of component-based systems. For example, researchers in formal methods have only recently begun to study principles of object-oriented software specification and verification, but do not yet have a good handle on how inheritance can be exploited in specification and verification. Other issues are also important in the practice of component-based systems, such as concurrency, mechanization and scalability, performance (time and space), reusability, and understandability. The aim is to brainstorm about these and related topics to understand both the problems involved and how formal techniques may be useful in solving them.

The goals of the workshop are to produce:

1. An outline of collaborative research topics,
2. A list of areas for further exploration,
3. An initial taxonomy of the different dimensions along which research in the area can be categorized. For instance, static/dynamic verification, modular/whole program analysis, partial/complete specification, soundness/completeness of the analysis, are all continuums along which particular techniques can be placed, and
4. A web site that will be maintained after the workshop to act as a central clearinghouse for research in this area.

We enthusiastically thank the authors of submitted papers; their quality contributions and participation are what make a workshop like SAVCBS successful. We thank the program committee for their careful reading and reviewing of the submissions. Our PC members have expertise in a wide variety of sub-disciplines related to specification and verification of component-based systems; they include established research leaders and promising recent Ph.D.s; they come from both industry and academia, and hail from all over the world.

We received 17 submissions. All papers were reviewed by at least 3 PC members. After PC discussions via a conference tool, 8 papers were accepted for long presentation at the workshop. Similar to previous years, we accepted 6 additional submissions for short presentation, reflecting the community-building role of SAVCBS and the goal of promoting discussion and incubation of new ideas for which a full paper may be premature. One of the accepted short presentations was withdrawn by the authors. Three submissions were rejected.

This year's program also includes a solution to a specification and verification challenge problem posed to workshop attendees. The problem focused on the specification of the subject-observer pattern. This common programming pattern is to separate the component that encapsulates some state from the components that access that state. The former component is often called a *subject*, while the latter type is an *observer*. At a minimum, a subject has a method for registering an observer, a method for updating the encapsulated state, and a method for retrieving the value of the state. Observers must provide a method for being notified: the behavior of the pair is that when the update method is called, all registered observers have their notification method called. While familiar to many programmers, this problem poses real challenges for specification and verification systems and it has already been the topic of a number of papers in the field. The received and presented solution was reviewed by two members of the program committee.

Arnd Poetzsch-Heffter (Program Committee Chair)

Jonathan Aldrich (Organizing Committee)
Mike Barnett (Organizing Committee)
Dimitra Giannakopoulou (Organizing Committee)
Gary T. Leavens (Organizing Committee)
Natasha Sharygina (Organizing Committee)

SAVCBS 2007 PAPERS

Effective Verification of Systems with a Dynamic Number of Components

Pavlína Vařeková[*] , Pavel Moravec[†] , Ivana Černá[*], Barbora Zimmerova[*]
Faculty of Informatics
Masaryk University
602 00 Brno, Czech Republic
{xvareko1, xmoravec, cerna, zimmerova}@fi.muni.cz

ABSTRACT

In the paper, we present a novel approach to verification of dynamic component-based systems, the systems that can have a changing number of components over their life-time. We focus our attention on systems with a stable part (called *provider*) and a number of dynamic components of one type (called *clients*) because dynamic systems can be often decomposed into segments like this. Our method for verification of such systems is based on determining a number k of dynamic components, such that if a system is proved correct for any number lower than k, it is consequently correct for an arbitrarily large number of dynamic components. The paper aims not only in proving the propositions that state this, it concentrates also on bounding the set of dynamic systems and verifiable properties in a way, that k is relatively small and thus practically interesting. In addition to this, we present an algorithm for computing k.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification

General Terms

Component-based systems, software verification

Keywords

Component-based systems, dynamic number of components, finite-state systems, formal verification

1. INTRODUCTION

The difficulty of formal verification of *dynamic component-based systems*, the systems that can have changing number of components over their life-time, is one of the most discussed issues these days in CBSE. It simply follows from the fact that if the number of components in a dynamic system is not bounded (it can grow over all bounds), the system model is infinite in general.

Our approach to the dynamic systems verification is based on two simple observations. First, the dynamic components[1] use to respect a *common type*, including a common template of their behaviour. Second, dynamic systems are often *not infinite by nature*. If we interpreted a dynamic system as a collection of its instances, each with a fixed number of dynamic components of each type, the instances would be finite-state, or at least have natural finite-state models. However their number would be infinite, which is the core of the problem.

In this paper, we focus on dynamic systems with a stable part (called *provider*) and a number of dynamic components of one type (called *clients*) because dynamic systems can be often decomposed into parts with this structure. The crucial idea of our approach to verification of a set of properties on such systems can be presented as follows. As the dynamic components share a common behaviour, there exists a number $k \in \mathbb{N}_0$ such that: *if the properties hold on the system with m dynamic components for any $m \leq k$, they also hold on the system with more then k dynamic components.* In the paper, we define such k, prove this statement true for it, and design an algorithm for computing it.

In particular, the value k for a dynamic system and a set of properties can be estimated as a sum of two measures. The first one is a measure of complexity of a dynamic system reflecting the maximal number of clients that are regarded by the provider. The second one is a similar measure on properties that reflects the minimal number of clients necessary to exhibit a path violating studied properties. In the paper, we focus on LTL-like properties, and we are interested only in those properties whose violation involves a limited number of components.

As an underlying formalism for this work, we use the *Component-interaction automata* language [11]. However the basic idea of this contribution is very general and the reasoning presented here is also applicable to other formalisms that

[*]The authors have been supported by the grant No. 1ET400300504.
[†]The author has been partially supported by the Grant Agency of Czech Republic grant No. 102/05/H050.

Sixth International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2007), September 3-4, 2007, Cavtat near Dubrovnik, Croatia.

[1]The components that are dynamically added and removed at run-time; analogy to instances of a type.

model systems as finite-state LTS or regular-like expressions.

The paper is structured as follows. First, we briefly discuss related work in Section 2. Then, Section 3 outlines the basic definitions of the Component-interaction automata language, and Section 4 formally defines dynamic systems that are of our interest. Section 5 designates the set of properties that are appropriate for verification using our approach, and it proves several propositions that are crucial for verification of dynamic systems. Section 6 is dedicated to the algorithm for finding the value k discussed above, and the text is closed with a conclusion and discussion of future work in Section 7.

2. RELATED WORK

As far as we are concerned, the idea presented above has not been elaborated yet. However there are other approaches that end up with finite models of potentially infinite dynamic systems. In [1], the author presents his solution to creating a finite model of a component that may communicate to arbitrary number of clients. It is assumed that even if generally the number of dynamic components connected to the component can be arbitrary, during the assembly phase, concrete number of the components is known. The author does not consider dynamic creation and removal of components at run-time. Another approach is studied in [10] where systems are first modelled as infinite-state and then reduced to finite-state by adjusted verification technique. In [7], the authors also use the technique of state-space reduction, in this case for verification of Java programs.

An alternative to state-space reduction is the verification of infinite-state models. This is discussed for instance in [2]. However these techniques are very time consuming and often do not guarantee to finish.

3. COMPONENT-INTERACTION AUTOMATA LANGUAGE

Component-interaction automata [5] are a specification language for modelling of component interactions in hierarchical component-based software systems. They capture each component as a labelled transition system with structured labels and a hierarchy of component names. The basic definitions are briefly reminded in this section. For more details, see [11].

Definition 3.1. A *hierarchy of component names*[2] is a tuple $H = (H_1, \ldots, H_m)$ of one of the following forms, where S_H denotes the set of component names corresponding to H. The first case is that $H_1, \ldots, H_m$ are pairwise different natural numbers or strings over the Greek alphabet (denoted $\mathbb{A}$ within the text); then $S_H = \bigcup_{i=1}^{m} \{H_i\}$. The second case is that $H_1, \ldots, H_m$ are hierarchies of component names where $S_{H_1}, \ldots, S_{H_m}$ are pairwise disjoint; then $S_H = \bigcup_{i=1}^{m} S_{H_i}$.

A *component-interaction automaton* (or a *CI automaton* for short) is a 5-tuple $\mathcal{C} = (Q, Act, \delta, I, H)$ where Q is a finite set of states, Act is a finite set of *actions*, $\Sigma = ((S_H \cup \{-\}) \times Act \times (S_H \cup \{-\})) \setminus (\{-\} \times Act \times \{-\})$ is a set of *labels*, $\delta \subseteq Q \times \Sigma \times Q$ is a finite set of *labelled transitions*, $I \subseteq Q$ is a nonempty set of *initial states*, and H is a hierarchy of component names.

The labels are triplets of a form $(-,a,o')$,$(o,a,-)$, or (o,a,o') and accordingly are of the type *input*, *output*, or *internal* respectively.
- The input label $(-,a,o')$ represents that the component o' receives the action a as an input.
- The output label $(o,a,-)$ represents that the component o sends the action a as an output.
- The internal label (o,a,o') represents that the component o sends the action a as an output and synchronously the component o' receives the action a as an input.

Examples of CI automata and their hierarchies of component names are in Figure 2 and Figure 4.

Definition 3.2. A *path* of a CI automaton $\mathcal{C} = (Q, Act, \delta, I, H)$ is an alternating sequence of states and labels given by δ that is either infinite, or is finite in case that it ends with a deadlock state. $Path(\mathcal{C})$ is the set of all paths of the CI automaton $\mathcal{C}$, $Path^{Init}(\mathcal{C})$ is its subset containing all paths starting in an initial state and $Path^{Init}_{Inf}(\mathcal{C})$ is the set of all infinite paths from $Path^{Init}(\mathcal{C})$.

If $\pi = q_0, l_0, q_1, l_1, \ldots$ is a (finite or infinite) path of the CI automaton $\mathcal{C}$, then
- $\mathcal{L}(\pi, i)$ is the i-th label of π (starting from $i = 0$) if there is any; here $\mathcal{L}(\pi, i) = l_i$,
- $\pi(i)$ is the i-th state of π if it exists; here $\pi(i) = q_i$ and
- π^i is the i-th suffix of π; here $\pi^i = q_i, l_i, \ldots$.

Notation. *For a given CI automaton $\mathcal{C}$ we denote $\mathcal{L}_\mathcal{C}$ the set of all labels reachable from an initial state in $\mathcal{C}$, $\mathcal{L}_{int,\mathcal{C}}$ the set of all internal labels reachable in $\mathcal{C}$.*

Definition 3.3. We say that a set of component-interaction automata $\{(Q_i, Act_i, \delta_i, I_i, H_i)\}_{i\in\mathcal{I}}$ is *composable* if $\mathcal{I} \subseteq \mathbb{N}$ is finite and $(H_i)_{i\in\mathcal{I}}$ is a hierarchy of component names.

Let $\mathcal{S} = \{(Q_i, Act_i, \delta_i, I_i, H_i)\}_{i\in\mathcal{I},\mathcal{I}\subseteq\mathbb{N}}$ be a finite composable set of component-interaction automata. Then the *complete transition space* for $\mathcal{S}$, written $\Delta_\mathcal{S}$, is a set of transitions among product states from $\Pi_{i\in\mathcal{I}} Q_i$ such that each transition reflects that either one of the automata from $\mathcal{S}$ follows its original transition and the others wait, or two automata synchronise on complementary labels $(o,a,-)$ and $(-,a,o')$, where $o, o' \in \mathbb{N} \cup \mathbb{A}$, and it forms a new label (o,a,o'). For precise definition see the appendix.

Let $\mathcal{S} = \{(Q_i, Act_i, \delta_i, I_i, H_i)\}_{i\in\mathcal{I}}$ be a composable set of CI automata and $\mathcal{F} \supseteq \bigcup_{i\in\mathcal{I}} \mathcal{L}_{int,\mathcal{C}_i}$ be a set of (feasible) labels. Then $\otimes^\mathcal{F}$ is a *composition operator with respect to feasible labels* which for the set $\mathcal{S}$ determines the CI automaton $\otimes^\mathcal{F}\mathcal{S} = (\Pi_{i\in\mathcal{I}} Q_i, \cup_{i\in\mathcal{I}} Act_i, \delta, \Pi_{i\in\mathcal{I}} I_i, (H_i)_{i\in\mathcal{I}})$ such that $\delta = \{(q, x, q') \in \Delta_\mathcal{S} \mid x \in \mathcal{F}\}$.

We say that the automaton $\otimes^\mathcal{F}\{\mathcal{C}_i\}_{i\in\mathcal{I}}$ is *defined* iff $\{\mathcal{C}_i\}_{i\in\mathcal{I}}$ is a composable set of CI automata and $\mathcal{F} \supseteq \bigcup_{i\in\mathcal{I}} \mathcal{L}_{int,\mathcal{C}_i}$.

Definition 3.4. Let $\mathcal{L}$ be a set of labels, S a set of component names. Then $Comm(\mathcal{L}, S)$ is a set of the labels $\mathcal{L}$ together with the internal labels that follow from $\mathcal{L}$ after communication with components whose names are in the set S.

[2]As distinct to the original definition, the component names do not need to be natural numbers, they can also be strings over Greek alphabet. This technicality does not influence any other definitions.

$$Comm(\mathcal{L},S) = \mathcal{L} \cup \{(o,a,o') \mid (o,a,-) \in \mathcal{L} \wedge o' \in S\} \cup \{(o,a,o') \mid (-,a,o') \in \mathcal{L} \wedge o \in S\}$$

Example 3.1. For $\mathcal{L} = \{(-,act_1,\alpha),(\beta,act_2,-),(\alpha,act_3,\beta)\}$ holds: $Comm(\mathcal{L},\{1,2\}) = \mathcal{L} \cup \{(1,act_1,\alpha),(\beta,act_2,1),(2,act_1,\alpha),(\beta,act_2,2)\}$.

3.1 The logic for specifying properties

In formal verification techniques, like *model checking* [6], the properties for verification are specified in temporal logics. In our approach, we use the logic CI-LTL [12]. CI-LTL is an extension of the action-based LTL [9], which is in addition able to express that a given action is enabled in a state of a path (one-step branching).

Definition 3.5. Let $\mathcal{L}$ be a set of labels of CI automata, then CI-LTL formulas over $\mathcal{L}$ are defined inductively:
1) If $l \in \mathcal{L}$, then $\mathcal{P}(l)$ and $\mathcal{E}(l)$ are formulas.
2) If ϕ and ψ are formulas, then $\phi \wedge \psi, \neg\,\phi, \mathcal{X}\,\phi$ and $\phi\ \mathcal{U}\ \psi$ are formulas.
3) Every formula can be obtained by a finite number of applications of previous two steps.

Let $\mathcal{C} = (Q, Act, \delta, I, H)$ be a CI automaton, then CI-LTL formulas are interpreted over the paths $\pi \in Path^{Init}_{Inf}(\mathcal{C})$ where the satisfaction relation $\models$ is defined inductively:

$$\begin{aligned}
\pi &\models \mathcal{E}(l) && \iff \exists q \in Q : \pi(0) \xrightarrow{l} q\\
\pi &\models \mathcal{P}(l) && \iff \mathcal{L}(\pi,0) = l\\
\pi &\models \varphi \wedge \psi && \iff \pi \models \varphi \text{ and } \pi \models \psi\\
\pi &\models \neg\,\varphi && \iff \pi \not\models \varphi\\
\pi &\models \mathcal{X}\,\varphi && \iff \pi^1 \models \varphi\\
\pi &\models \varphi\ \mathcal{U}\ \psi && \iff \exists j \in \mathbb{N}_0 : \pi^j \models \psi \text{ and}\\
& && \quad \forall k \in \mathbb{N}_0, k < j : \pi^k \models \phi
\end{aligned}$$

Other operators can be defined as shortcuts: $\varphi \vee \psi \equiv \neg\,(\neg\,\varphi \wedge \neg\,\psi)$, $\varphi \Rightarrow \psi \equiv \neg\,(\varphi \wedge \neg\,\psi)$, $\mathcal{F}\,\varphi \equiv \text{true}\ \mathcal{U}\ \varphi$, $\mathcal{G}\,\varphi \equiv \neg\,\mathcal{F}\,\neg\,\varphi$.

Notation. *Let φ be a CI-LTL formula then $\mathcal{L}_\varphi$ is a set of labels that occur in the formula.*

4. DYNAMIC SYSTEM MODEL

To simplify the explication of our approach, we narrow our attention to the dynamic systems consisting of a number of dynamic clients which are connected to one provider that represents the stable part of the system. We assume that the clients have the same behaviour, and that they do not communicate with each other (illustrated in Figure 1). This is natural in any system where the clients may be added and removed dynamically.

Additionally, we focus only on the systems that fulfil that: (1) the system has only one type of clients, and (2) each client is modelled as an automaton with the hierarchy of component names (i). Note that the previous conditions do not significantly restrict the number of systems we can deal with. Renaming and system partitioning usually helps to transform a system into the permissible one.

In the definition below, the description is reflected as follows. The first bullet states that the components constituting the provider are named with strings over Greek alphabet.

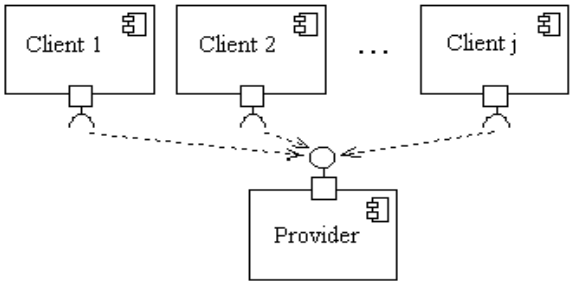


Figure 1: A dynamic system with j clients.

The second bullet reflects that all the clients have the same models up to the names of modelled components, where i-th client models a component with a numerical name i. The third, fourth and fifth bullet restrict the transitions allowed in the composition of the automata. The third bullet states that no two clients may communicate with each other, the fourth assures that all clients are handled equally by the provider, and the fifth care for the composition to be defined.

Definition 4.1. Let $\mathcal{C}_0 = (Q_0, Act_0, \delta_0, I_0, H_0)$ and $\mathcal{C}_i = (Q, Act, \delta_i, I, (i))$, where $i \in \mathbb{N}$, be CI automata and $\mathcal{F}$ be a set of labels. The tuple $(\mathcal{C}_0, \{\mathcal{C}_i\}_{i\in\mathbb{N}}, \mathcal{F})$ is a *CI model of a dynamic system* (or a *dynamic system model* for short) iff:
- $S_{H_0} \subseteq \mathbb{A}$,[3]
- for all $i \in \mathbb{N}$ the CI automaton, which results from $\mathcal{C}_i$ after renaming of all component names with $r : \{i\} \to \{1\}$, is equal to the CI automaton $\mathcal{C}_1$,
- $\mathcal{F} \cap \{(i,act,j) \mid i,j \in \mathbb{N}, i \neq j\} = \emptyset$,
- each permutation p of the set $\mathbb{N} \cup \mathbb{A} \cup \{-\}$, which is the identity on the set $\mathbb{A} \cup \{-\}$ fulfils:
$$\mathcal{F} = \{(p(o_1),a,p(o_2)) \mid (o_1,a,o_2) \in \mathcal{F}\},$$
- $\mathcal{F} \supseteq \mathcal{L}_{int,\mathcal{C}_0} \cup \bigcup_{i\in\mathbb{N}} \mathcal{L}_{int,\mathcal{C}_i}$.

Notation. *For a dynamic system model $\mathcal{D} = (\mathcal{C}_0, \{\mathcal{C}_i\}_{i\in\mathbb{N}}, \mathcal{F})$ the automaton $\mathcal{C}_0$ is called* provider*, CI automata $\mathcal{C}_1, \mathcal{C}_2, \ldots$ are called* clients.

For the rest of the paper let us fix that if $\mathcal{D}$ is a dynamic system, then it denotes the tuple:
$(\mathcal{C}_0 = (Q_0, Act_0, \delta_0, I_0, H_0), \{\mathcal{C}_i = (Q, Act, \delta_i, I, (i))\}_{i\in\mathbb{N}}, \mathcal{F})$.

Definition 4.2. Let $\mathcal{D}$ be a dynamic system and $n \in \mathbb{N}_0$, then $\mathcal{D}_n = \otimes^{\mathcal{F}}\{\mathcal{C}_i\}_{0\le i\le n}$ is the CI automaton modelling system $\mathcal{D}$ with n clients and $\mathcal{L}_\mathcal{D} = \bigcup_{i\in\mathbb{N}_0} \mathcal{L}_{\mathcal{D}_i}$ is the set of all labels reachable in any of the automata $\{\mathcal{D}_i\}_{i\in\mathbb{N}_0}$.

Example 4.1. Let us consider a simple example of a dynamic system model depicted in Figure 2 capturing a simple system consisting of a database and its clients.

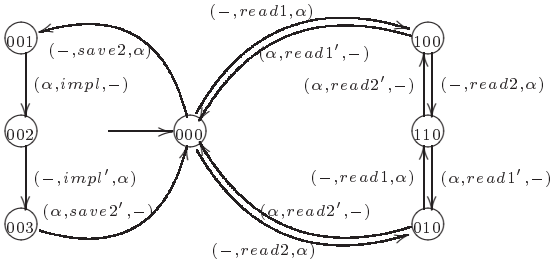
The database (modelled by the component $\mathcal{C}_0$) provides three types of services: $read1$, $read2$ and $save2$ which correspond to standard reading or saving of data in a database. Each of the services is modelled by a tuple of actions, the first of them, e.g. $(-,read1,\alpha)$, represents a receiving of a request of the service and the second, e.g. $(\alpha,read1',-)$, models the response, which indicates that the service was finished. Services $read1$ and $read2$ can be executed in parallel, but $save2$

[3] $\mathbb{A}$ is the set of strings over Greek alphabet

can not. Because of space reasons, services $read1$ and $read2$ are modelled without implementation details. Service $save2$ is modelled as a service that can be provided only after a connection of a component that implements the service.

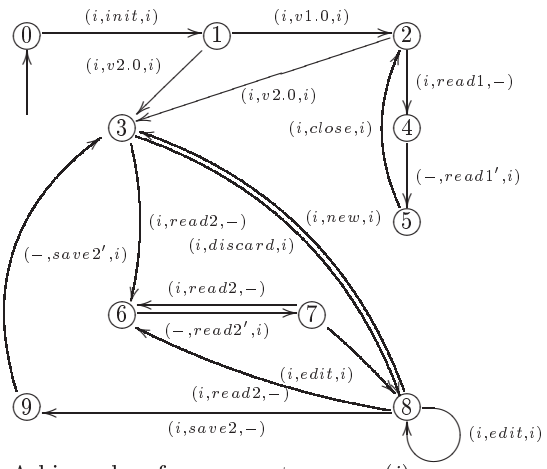
Clients (modelled by components $\mathcal{C}_i$, $i \in \mathbb{N}$) can use three previously mentioned types of services. They first initialise their system and after that they choose one of two versions of the software. If they choose version 1.0, they can only read data from the database or upgrade their software, else they can read data, edit them and save the changes.

- $\mathcal{C}_0$:



A hierarchy of component names: (α)

- $\mathcal{C}_i$, where $i \in \mathbb{N}$:



A hierarchy of component names: (i)

- $\mathcal{F} = \bigcup_{i\in\mathbb{N}} \mathcal{L}_{int,\mathcal{C}_i} \cup \{(\alpha,impl,-),(-,impl',\alpha)\} \cup \{(i,read2,\alpha),(\alpha,read2',i),(i,save2,\alpha),(\alpha,save2',i),(i,read1,\alpha),(\alpha,read1',i) \mid i \in \mathbb{N}\}$.

Figure 2: A CI model of dynamic system $\mathcal{DB} = (\mathcal{C}_0, \{\mathcal{C}_i\}_{i\in\mathbb{N}}, \mathcal{F})$.

5. VERIFICATION

In this section, we study verification of the properties whose violation involves at most a finite number of clients. We denote m the minimal number of clients that are necessary for the violation. For an arbitrary integer m and a dynamic system model $\mathcal{D}$ the set of such properties is denoted $Property(\mathcal{D}, m)$. A property for $m = 2$ can for example state that *whenever a dynamic component outputs a request for service a, no other dynamic component gets the response for a before the previous component does.* If the dynamic components behave the same and do not communicate with each other, it really suffices to observe only two of them to check this property.

In this section, we first discuss when we can say that a provider *regards at most n* clients. After definition of this term, we show that if a provider in $\mathcal{D}$ regards at most n clients, then given a set of properties $Property(\mathcal{D}, m)$ it holds that if we verify all of these on the dynamic system with $0, 1, \ldots, n+m$ clients, then the properties hold on the dynamic system *with any finite number of clients.*

5.1 Essential definitions and lemmas

For each dynamic system model $\mathcal{D}$ and each set of labels $X \subseteq \mathcal{L}_{\mathcal{C}_0}$ we define the value n from the previous paragraph. Then we prove some basic properties that the value n fulfils.

The sub-section starts with the definition of $\pi_{\Delta,X}$, which is in fact an abstraction of the path π pointing up important parts of the path given by the set of labels X. It is followed by the essential definition stating when we can say that a provider regards at most n clients.

Definition 5.1. Let $\mathcal{D}$ be a dynamic system model, $X \subseteq \mathcal{L}_{\mathcal{C}_0}$, $j \in \mathbb{N}$ and $\pi = q_0, l_0, q_1, l_1, q_2, \cdots \in Path^{Init}_{Inf}(\mathcal{D}_j)$. Then $\pi_{\Delta,X} = h(q_0,l_0,q_1), h(q_1,l_1,q_2), \ldots,$ such that h is defined as:

$$h\big((r_0,\ldots,r_j),(o,a,o'),(r'_0,\ldots,r'_j)\big) = \begin{cases} r_0,(f(o),a,f(o')),r'_0 : \\ \quad (o,a,o') \in Comm(X,\mathbb{N}) \\ \epsilon : \quad \text{otherwise,} \end{cases}$$

where ϵ is an empty string and $f(o) = \begin{cases} o, & o \notin \mathbb{N}, \\ *, & o \in \mathbb{N}. \end{cases}$

Example 5.1. For the dynamic system model $\mathcal{DB}$ described in Figure 2, $X = \{(-,read2,\alpha),(\alpha,read2',-)\}$, $j = 4$ and

$$\begin{aligned} \pi = (000,0,0,0) &\overset{(3,init,3)}{\rightarrow} (000,0,0,1) \overset{(3,v2.0,3)}{\rightarrow} \\ &\rightarrow (000,0,0,3) \overset{(3,read2,\alpha)}{\rightarrow} (010,0,0,6) \overset{(1,init,1)}{\rightarrow} \\ &\rightarrow (010,1,0,6) \overset{(1,v1.0,1)}{\rightarrow} (010,2,0,6) \overset{(1,read1,\alpha)}{\rightarrow} \\ &\rightarrow (011,4,0,6) \overset{(\alpha,read2',3)}{\rightarrow} (001,4,0,7) \cdots \end{aligned}$$

it holds: $\pi_{\Delta,X} = \boxed{000} \xrightarrow{(*,read2,\alpha)} \boxed{010}, \boxed{011} \xrightarrow{(\alpha,read2',*)} \boxed{001}, \cdots$

Definition 5.2. Let $\mathcal{D}$ be a dynamic system model, $X \subseteq \mathcal{L}_{\mathcal{C}_0}$ contains all observable labels, and $n \in \mathbb{N}$. We say that *the provider at any time regards at most n clients* iff for any $i \geq n$ it holds that:
$\{\pi_{\Delta,X} \mid \pi \in Path^{Init}_{Inf}(\mathcal{D}_i)\} = \{\pi_{\Delta,X} \mid \pi \in Path^{Init}_{Inf}(\mathcal{D}_n)\}$.

Notation. *Let $\mathcal{D}$ be a dynamic system model and $X \subseteq \mathcal{L}_{\mathcal{C}_0}$ again contains all observable labels. Then $|\mathcal{D}|_X$ is a minimal n such that the provider at any time regards at most n clients, if there is any. If there is no such n then $|\mathcal{D}|_X = \infty$.*

Example 5.2. For the dynamic system model $\mathcal{DB}$ described in Figure 2 and
$X = \{(-,save2,\alpha),(\alpha,impl,-),(-,impl',\alpha),(\alpha,save2',-)\}$
it holds that:

- $\{\pi_{\Delta,X} \mid \pi \in Path^{Init}_{Inf}(\mathcal{DB}_0)\} = \emptyset$,
- $\{\pi_{\Delta,X} \mid \pi \in Path^{Init}_{Inf}(\mathcal{DB}_1)\}$ contains exactly one infinite sequence σ of the shape:

  $\boxed{000} \xrightarrow{(*,save2,\alpha)} \boxed{001}, \boxed{001} \xrightarrow{(\alpha,impl,-)} \boxed{002}, \boxed{002} \xrightarrow{(-,impl',\alpha)} \boxed{003},$
  $\boxed{003} \xrightarrow{(\alpha,save2',*)} \boxed{000}, \boxed{000} \xrightarrow{(*,save2,\alpha)} \boxed{001}, \cdots$

  The set moreover contains all finite prefixes of σ ending with "$\boxed{003} \xrightarrow{(\alpha,save2',*)} \boxed{000}$" and the empty sequence ϵ.

- $\{\pi_{\Delta,X} \mid \pi \in Path^{Init}_{Inf}(\mathcal{D}B_2)\} =$
  $\{\pi_{\Delta,X} \mid \pi \in Path^{Init}_{Inf}(\mathcal{D}B_1)\}$.

Hence the database regards at most one client.

The following lemma shows, that if in a dynamic system model $\mathcal{D}$ we delete the transitions over any subset of observable labels $Comm(X,\mathbb{N})$, such that the result $\mathcal{D}'$ is a dynamic system model, then it fulfils $|\mathcal{D}'|_X \leq |\mathcal{D}|_X$.

Lemma 5.1. *Let $\mathcal{D}$ be a dynamic system model, $X \subseteq \mathcal{L}_{\mathcal{C}_0}$. If a dynamic system model $\mathcal{D}' = (\mathcal{C}_0, \{\mathcal{C}_i\}_{i\in\mathbb{N}}, \mathcal{F} \setminus S)$ fulfils $S \subseteq Comm(X,\mathbb{N})$, then $|\mathcal{D}'|_X \leq |\mathcal{D}|_X$.*

Proof. The statement follows immediately from the fact that all deleted transitions, which occur in automata $\{\mathcal{D}_i\}_{i\in\mathbb{N}_0}$ and do not occur in automata $\{\mathcal{D}'_i\}_{i\in\mathbb{N}_0}$, have labels in the set $Comm(X,\mathbb{N})$. □

The next lemma involves dynamic system models, in which providers are composite components such that one of their sub-components contains all actions that are used for communication with clients. This lemma can be used for finding an over-approximation of the value $|\mathcal{D}|_X$ for these systems. In addition to these, it has a corollary useful for verification of properties which we are interested in.

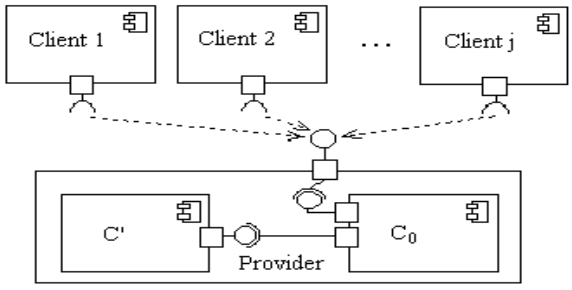


Figure 3: A dynamic system with composed component according to Lemma 5.2 and j clients.

Lemma 5.2. *Let $\mathcal{D}$ be a dynamic system model and $X \subseteq \mathcal{L}_{\mathcal{C}_0}$. Provided that $\mathcal{C}' = (Q', Act', \delta', I', H')$ is a CI automaton and $\mathcal{F}'$ is a set of labels such that $\otimes^{\mathcal{F}'}\{\mathcal{C}_0, \mathcal{C}'\}$ is defined, $S_{H'} \subseteq \mathbb{A}$, then $\mathcal{D}' = (\otimes^{\mathcal{F}'}\{\mathcal{C}_0, \mathcal{C}'\}, \{\mathcal{C}_i\}_{i\in\mathbb{N}}, \mathcal{F})$ is a dynamic system model. Further, let the following holds:*

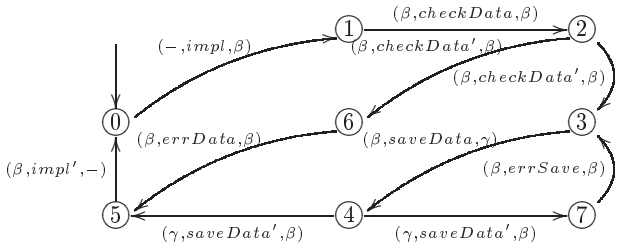
1) $(o',a,o) \in \mathcal{F}', o' \in S_{H'} \Rightarrow o \in \{-\} \cup S_{H_0} \cup S_{H'}$,
   $(o,a,o') \in \mathcal{F}', o' \in S_{H'} \Rightarrow o \in \{-\} \cup S_{H_0} \cup S_{H'}$,
2) $(o,a,o') \in \mathcal{F}', o' \in S_{H'}, o \in S_{H_0} \Rightarrow (o,a,-) \in X \cap \mathcal{F}$,
   $(o',a,o) \in \mathcal{F}', o' \in S_{H'}, o \in S_{H_0} \Rightarrow (-,a,o) \in X \cap \mathcal{F}$,
3) $\forall i \geq |\mathcal{D}|_X : Path^{Init}(\mathcal{D}_i) = Path^{Init}_{Inf}(\mathcal{D}_i)$.

Then for $X' \subseteq Comm(X, S_{H'}) \cup \mathcal{L}_{\mathcal{C}'}$ is $|\mathcal{D}'|_{X'} \leq |\mathcal{D}|_X$.

Proof. See appendix.

Example 5.3. This example illustrates a usage of previous two lemmas. Let us consider the dynamic system model $\mathcal{D}B$ from Figure 2. In Figure 4 there is a CI automaton $\mathcal{C}'$ which implements service *save*, thus the CI automaton $C_0^{impl} = \otimes^{\mathcal{F}'}\{\mathcal{C}_0, \mathcal{C}'\}$ models the database where all the services are implemented, and $\mathcal{D}B^{impl} = (C_0^{impl}, \{\mathcal{C}_i\}_{i\in\mathbb{N}}, \mathcal{F}''')$, where $\mathcal{F}'$, $\mathcal{F}'''$ are described in Figure 4, models a dynamic

- $\mathcal{C}'$:



A hierarchy of component names: (β, γ)

- $\mathcal{F}' = \mathcal{L}_{\mathcal{C}_0} \cup \mathcal{L}_{\mathcal{C}'} \cup \{_{(\alpha,impl,\beta),(\beta,impl',\alpha)}\}$
- $\mathcal{F}'' = \mathcal{F} \cup \mathcal{L}_{int,\mathcal{C}_0^{impl}}$
- $\mathcal{F}''' = \mathcal{F}'' \setminus \{_{(\alpha,impl,-),(-,impl',\alpha)}\}$.

Figure 4: A CI automaton $\mathcal{C}'$ and sets $\mathcal{F}'$, $\mathcal{F}''$, $\mathcal{F}'''$.

system whose provider is a database with all services implemented. Its clients are the same as in database $\mathcal{D}B$ and the clients can use the services of the provider similarly as in $\mathcal{D}B$. Lemma 5.2 says that for all $X \supseteq \{_{(\alpha,impl,-),(-,impl',\alpha)}\}$ and $X' \subseteq Comm(X,\mathbb{N})$:

$$|(\mathcal{C}_0^{impl}, \{\mathcal{C}_i\}_{i\in\mathbb{N}}, \mathcal{F}'')|_{X'} \leq |DB|_X,$$

where $\mathcal{F}''$ is described in Figure 4, and Lemma 5.1 then claims that for those X it in addition holds that:
$|DB^{impl}|_X = |(\mathcal{C}_0^{impl}, \{\mathcal{C}_i\}_{i\in\mathbb{N}}, \mathcal{F}''')|_{X'} \leq |(\mathcal{C}_0^{impl}, \{\mathcal{C}_i\}_{i\in\mathbb{N}}, \mathcal{F}'')|_{X'}$.

5.2 Properties $Property(\mathcal{D}, m)$

Above, for each dynamic system $\mathcal{D}$ and a set of observable labels X, we have defined the value $|\mathcal{D}|_X$. In this part, we show the employment of the value in verification of dynamic systems.

Assume a dynamic system $\mathcal{D}$. The properties that we aim to verify, can be specified with a sequence of formulas $\{\varphi_i\}_{i\in\mathbb{N}_0}$ over $\mathcal{L}_{\mathcal{D}}$ such that a property is satisfied iff for each $i \in \mathbb{N}_0$ it holds that $\varphi_i \models \mathcal{D}_i$. Note that not every sequence of formulas $\{\varphi_i\}_{i\in\mathbb{N}_0}$ represents a meaningful property of the system. Thus we concentrate on the formulas satisfying:
- the property makes no distinctions among clients,
- if the property is violated by a path in a system $\mathcal{D}_{i+j}$ where j components do not perform any steps, the property is violated by the same sequence of labels also in the system with i clients only.

Definition 5.3. Let $\mathcal{D}$ be a dynamic system model. A sequence $\{\varphi_i\}_{i\in\mathbb{N}}$ of formulas over the set of labels $\mathcal{L}_{\mathcal{D}}$ is *harmonised with the dynamic system* $\mathcal{D}$ iff for all $i \in \mathbb{N}$:
- formula φ_i and formula φ_i with permutated numerical names of components have the same set of models,
- let
$\pi^i = (q_0^0, q_0^1, \ldots, q_0^i), l_0, (q_1^0, q_1^1, \ldots, q_1^i), l_1, \ldots \in Path^{Init}_{Inf}(\mathcal{D}_i)$
and for $j \in \mathbb{N}$, $\pi^{i+j} =$
$(q_0^0, \ldots, q_0^i, q_0^{i+1}, \ldots, q_0^{i+j}), l_0, (q_1^0, \ldots, q_1^i, q_0^{i+1}, \ldots, q_0^{i+j}), \ldots$
in $Path^{Init}_{Inf}(\mathcal{D}_{i+j})$ then $\pi^{i+j} \not\models \varphi_{i+j}$ implies $\pi^i \not\models \varphi_i$.

As we have discussed before, we are not interested in all formula sequences implied by the definition above. We focus only on the formula sequences that represent properties whose violation involves only a finite number of observed components. Moreover, we consider only the properties that

are invariant under stuttering according to CI-LTL. The first restriction is requisite, but the second one is not. If we did not apply the second restriction, we would in many cases end up with $|\mathcal{D}|_X = \infty$.

Definition 5.4. Two paths π and σ are *stuttering equivalent* wrt. a set of labels $\mathcal{L}$ iff there are two finite or infinite sequences $0 = i_0^1 < i_1^1 < \cdots$ and $0 = i_0^2 < i_1^2 < \cdots$ such that for each index $j \geq 1$ and each $l \in \mathcal{L}$, the following holds:

- $\pi^{i_{j-1}^1} \models \mathcal{E}(l) \Leftrightarrow \pi^{i_{j-1}^1+1} \models \mathcal{E}(l) \Leftrightarrow \cdots \Leftrightarrow \pi^{i_j^1-1} \models \mathcal{E}(l)$
  $\Longleftrightarrow$
  $\sigma^{i_{j-1}^2} \models \mathcal{E}(l) \Leftrightarrow \sigma^{i_{j-1}^2+1} \models \mathcal{E}(l) \Leftrightarrow \cdots \Leftrightarrow \sigma^{i_j^2-1} \models \mathcal{E}(l)$
- $\pi^{i_j^1-1} \models \mathcal{P}(l) \Longleftrightarrow \sigma^{i_j^2-1} \models \mathcal{P}(l)$
- $\forall l \in \mathcal{L}, i^1 \in \mathbb{N}: \ \pi^{i^1} \models \mathcal{P}(l) \Rightarrow i^1 = i_j^1 - 1 \ (j \in \mathbb{N})$
- $\forall l \in \mathcal{L}, i^2 \in \mathbb{N}: \ \sigma^{i^2} \models \mathcal{P}(l) \Rightarrow i^2 = i_j^2 - 1 \ (j \in \mathbb{N})$.

Observe that if π and σ are stuttering equivalent paths, then for each $j \geq 0$, all the paths $\pi^{i_j^1}, \ldots, \pi^{i_{j+1}^1-1}$ and $\sigma^{i_j^2}, \ldots, \sigma^{i_{j+1}^2-1}$ are also pairwise stuttering equivalent.

Definition 5.5. A CI-LTL formula φ is called *invariant under stuttering* iff for each two paths π and σ stuttering equivalent wrt. $\mathcal{L}_\varphi$, the equivalence $\pi \models \varphi \Leftrightarrow \sigma \models \varphi$ holds.

Lemma 5.3. *Let φ be a CI-LTL formula which does not contain operator $\mathcal{X}$ and any occurrence of an atomic proposition $\mathcal{P}(l)$ in φ is of the form either $\phi_1 \mathcal{U}(\mathcal{P}(l) \wedge \phi_2)$ or $(\phi_1 \wedge \neg\mathcal{P}(l))\mathcal{U}\phi_2$ or $(\phi_1 \wedge \neg\mathcal{P}(l_1))\mathcal{U}(\mathcal{P}(l_2) \wedge \phi_2)$. Then φ is invariant under stuttering.*

Proof. See appendix.

Definition 5.6. Let $\mathcal{D}$ be a dynamic system model and $\{\varphi_i\}_{i\in\mathbb{N}}$ be a sequence of formulas harmonised with $\mathcal{D}$. Then we define $|\{\varphi_i\}_{i\in\mathbb{N}}|_\mathcal{D}$ as the minimal $m \in \mathbb{N}_0$ such that for each $j \in \mathbb{N}$ and each path $\pi \in Path_{Inf}^{Init}(\mathcal{D}_j)$ satisfying $\pi \not\models \varphi_j$ there is an invariant under stuttering subformula $\varphi_{j,\pi}$ of formula φ_j and there are names of components $i_1, \ldots, i_m \in \mathbb{N}$ that fulfil

- $\pi \not\models \varphi_{j,\pi}$,
- formula $\neg\varphi_{j,\pi} \Rightarrow \neg\varphi_j$ is satisfied on each automaton, whose set of labels is a superset of $\mathcal{L}_{\varphi_{j,\pi}} \cup \mathcal{L}_{\varphi_j}$,
- $\mathcal{L}_{\varphi_{j,\pi}} \subseteq \mathcal{L}_\mathcal{D}$ and $\mathcal{L}_{\varphi_{j,\pi}}$ does not contain the labels that involve names of dynamic components that are different from $i_1, \ldots, i_m$.

Definition 5.7. For a dynamic system model $\mathcal{D}$ and $m \in \mathbb{N}_0$, $Property(\mathcal{D}, m)$ is the set of all sequences harmonised with the dynamic system $\mathcal{D}$ such that

$$\{\varphi_i\}_{i\in\mathbb{N}} \in Property(\mathcal{D}, m) \text{ iff } |\{\varphi_i\}_{i\in\mathbb{N}}|_\mathcal{D} \leq m.$$

Example 5.4. In this example, the previous definition is illustrated by three properties of dynamic system model $\mathcal{DB}$ from Figure 2 described by a sequence of CI-LTL formulas.

1) Consider the sequence of formulas
$\varphi_0 = \varphi_1 = true$
$\varphi_2 = \phi(1,2)$
$\varphi_3 = \phi(1,2) \wedge \phi(1,3) \wedge \phi(2,3)$,
$\ldots$,
where
$\phi(i,j) = \neg\mathcal{F}\left(\mathcal{E}_{(i,save2',\alpha)} \wedge \mathcal{E}_{(j,save2',\alpha)}\right)$
capturing that *a state, in which the provider can respond that the action 'save' was done to two clients, is unreachable.* This sequence is in $Property(\mathcal{DB}, 2)$ because for arbitrary $n \in \mathbb{N}_0$ and $\pi \in Path_{Inf}^{Init}(\mathcal{DB}_n)$ satisfying $\pi \not\models \varphi_n$, there exists $i \in \mathbb{N}_0$ such that for different numbers $j_1, j_2 \in \mathbb{N}$ the actions $_{(j_1,save2',\alpha)}$, $_{(j_2,save2',\alpha)}$ are enabled in the state $\pi(i)$. Thus $\pi \not\models \phi(j_1, j_2) = \varphi_{n,\pi}$ and it is obvious that the formula $\varphi_{n,\pi}$ fulfils all conditions in Definition 5.6.

2) The set $Property(\mathcal{DB}, 0)$ contains all temporal properties described by a sequence of identical formulas $\{\varphi_i = \varphi\}_{i\in\mathbb{N}_0}$, where φ is invariant under stuttering and $\mathcal{L}_\varphi$ does not contain labels that involve any clients.
For example:
$\varphi = \mathcal{G}\left(\mathcal{P}_{(\alpha,impl,-)} \Rightarrow \mathcal{F}\,\mathcal{P}_{(-,impl',\alpha)}\right)$.
This sequence of formulas is in $Property(\mathcal{DB}, 0)$ because for an arbitrary $n \in \mathbb{N}_0$ and $\pi \in Path_{Inf}^{Init}(\mathcal{DB}_n)$ satisfying $\pi \not\models \varphi_n$ the formula $\varphi_{n,\pi} = \varphi_n$ fulfils all conditions in Definition 5.6.

3) The set $Property(\mathcal{DB}, 1)$ contains the sequence of formulas
$\varphi_0 = true$
$\varphi_1 = \phi(1)$,
$\varphi_2 = \phi(1) \wedge \phi(2)$,
$\varphi_3 = \phi(1) \wedge \phi(2) \wedge \phi(3)$,
$\ldots$,
where
$\phi(i) = \mathcal{G}\left(\mathcal{P}_{(i,edit,i)} \Rightarrow \mathcal{F}\,\mathcal{E}_{(i,save,\alpha)}\right)$,
capturing that *globally after some component edits an entry in the database, it will be eventually enabled to save the changes.*

Similarly as in the case 1) it can be shown that this sequence of formulas is in the set $Property(\mathcal{DB}, 1)$.

Finally, the following lemma claims that for verification of the properties from the set $Property(\mathcal{D}, 0)$ it suffices to verify the models $\mathcal{D}_0, \ldots, \mathcal{D}_{|\mathcal{D}|_X}$. The theorem below formulates an analogical result for the properties $Property(\mathcal{D}, m)$, $m \in \mathbb{N}$, stating that we only need to verify the models $\mathcal{D}_0, \ldots, \mathcal{D}_{|\mathcal{D}|_X+m}$. The idea of this theorem is based on the modification of the dynamic system $\mathcal{D}$ and the corresponding modification of verified temporal property. To these modified system and property, we can apply Lemma 5.2 and get the statement from the theorem. [4]

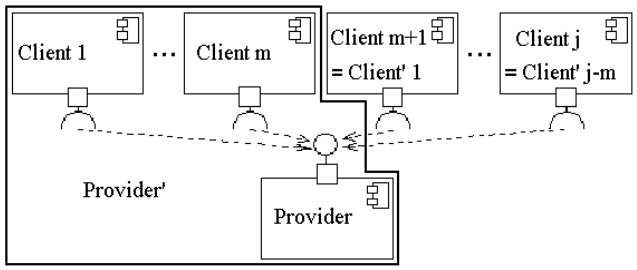


Figure 5: A dynamic system $\mathcal{D}$ with j clients and dynamic system $\mathcal{D}'$ with $j - m$ clients.

[4] The modified model $\mathcal{D}'$ is created from $\mathcal{D}$ by modification of its provider, modelling the provider of $\mathcal{D}$ composed with m clients. The remaining items of the dynamic system are identical to $\mathcal{D}$ (see Figure 5). The modification of property is more complex and it is described in detail in the proof of the theorem.

Definition 5.8. Let $\mathcal{D}$ be a dynamic system model and $\{\varphi_i\}_{i\in\mathbb{N}}$ then a set of labels X *contains all labels necessary for verification of* $\{\varphi_i\}_{i\in\mathbb{N}}$ *on* $\mathcal{D}$ iff

- $\mathcal{P}(l)$ is a subformula of φ_i for some $i \in \mathbb{N}_0 \Rightarrow$ $l \in Comm(X, \mathbb{N})$
- $\mathcal{E}(l)$ is a subformula of φ_i for some $i \in \mathbb{N}_0 \Rightarrow$ $\forall$ transition (q, l', q') in $\mathcal{D}_i$ iff $q \models \mathcal{E}(l)$ and $q' \not\models \mathcal{E}(l)$ or vice versa, then $l \in Comm(X, \mathbb{N})$.

Lemma 5.4. *Let* $\mathcal{D}$ *be a dynamic system model and* $\{\varphi_i\}_{i\in\mathbb{N}} \in Property(\mathcal{D}, 0)$ *and* X *contains all necessary labels for verification of it. Then for every* $j \in \mathbb{N}$ *it holds that:*

$$\mathcal{D}_{|\mathcal{D}|_X} \models \varphi_{|\mathcal{D}|_X} \;\Rightarrow\; \mathcal{D}_{|\mathcal{D}|_X+j} \models \varphi_{|\mathcal{D}|_X+j}.$$

Proof. See appendix.

Theorem 5.1. *Let* $\mathcal{D}$ *be a dynamic system model,* $\{\varphi_i\}_{i\in\mathbb{N}} \in Property(\mathcal{D}, m)$, X *contains all necessary labels for verification of it and*

- $\forall i \geq |\mathcal{D}|_X$: $Path^{Init}(\mathcal{D}_i) = Path^{Init}_{Inf}(\mathcal{D}_i)$,
- $\big((a,act,n) \in \mathcal{F}$ *for some* $n \in \mathbb{N}\big) \;\Rightarrow\; (a,act,-) \in X \cap \mathcal{F}$,
- $\big((n,act,a) \in \mathcal{F}$ *for some* $n \in \mathbb{N}\big) \;\Rightarrow\; (-,act,a) \in X \cap \mathcal{F}$.

Then for every $j \in \mathbb{N}$ *it holds that:*

$$\mathcal{D}_{|\mathcal{D}|_X+m} \models \varphi_{|\mathcal{D}|_X+m} \;\Rightarrow\; \mathcal{D}_{|\mathcal{D}|_X+m+j} \models \varphi_{|\mathcal{D}|_X+m+j}.$$

Proof. See appendix.

6. ALGORITHM

In the previous section, we have shown that if for a dynamic system model $\mathcal{D}$, a property $\{\varphi_i\}_{i\in\mathbb{N}}$ and a set X that contains all labels necessary for verification of $\{\varphi_i\}_{i\in\mathbb{N}}$ on $\mathcal{D}$, we know the value m such that $\{\varphi_i\}_{i\in\mathbb{N}} \in Property(\mathcal{D}, m)$ and an over-approximation n of the value $|\mathcal{D}|_X$, then we can verify the property via verification of the models $\mathcal{D}_0, \ldots, \mathcal{D}_{m+n}$. We did not discuss how we can find appropriate X, m and n, which we are going to do now. The set X can be constructed automatically based on Definition 5.8. The value m such that $\{\varphi_i\}_{i\in\mathbb{N}} \in Property(\mathcal{D}, m)$ can be derived from the structure of the formulas, and hence we can suppose that this value is known already at the time when the formulas for a given property are constructed. The computation of a reasonable over-approximation of the value $|\mathcal{D}|_X$ does not need to be so straightforward. Hence in this section, we discuss the technique that allows us to compute the over-approximation of $|\mathcal{D}|_X$ for most of the dynamic system models $\mathcal{D}$ such that $|\mathcal{D}|_X \neq \infty$.

For this purpose, we employ the value $\|\mathcal{D}\|_X$, which reflects the number of clients that the provider can serve concurrently at any moment. The value $\|\mathcal{D}\|_X$ can be computed automatically and it always holds that $\|\mathcal{D}\|_X \leq |\mathcal{D}|_X$. For the most common type of dynamic systems where $|\mathcal{D}|_X < \infty$, there exists a computable value z such that $|\mathcal{D}|_X \leq 1 + \|\mathcal{D}\|_X \cdot z$.

6.1 Algorithm for computing $\|\mathcal{D}\|_X$

We first define the value $\|\mathcal{D}\|_X$ and then prove that $\|\mathcal{D}\|_X \leq |\mathcal{D}|_X$ always holds.

Definition 6.1. Let $\mathcal{D}$ be a dynamic system model, $X \subseteq \mathcal{C}_0$. Then a state $q \in Q$ is *not in a cycle of service* X iff it is in a set $N_{\mathcal{D},X} \subseteq Q$ defined inductively:

- $I \subseteq N_{\mathcal{D},X}$,
- $(q \in N_{\mathcal{D},X} \wedge \exists l \notin Comm(X,\mathbb{N}) \colon (q,l,q') \in \delta_1) \;\Rightarrow\; q' \in N_{\mathcal{D},X}$,
- $(q \in N_{\mathcal{D},X} \wedge \exists l \notin Comm(X,\mathbb{N}) \colon (q',l,q) \in \delta_1) \;\Rightarrow\; q' \in N_{\mathcal{D},X}$.

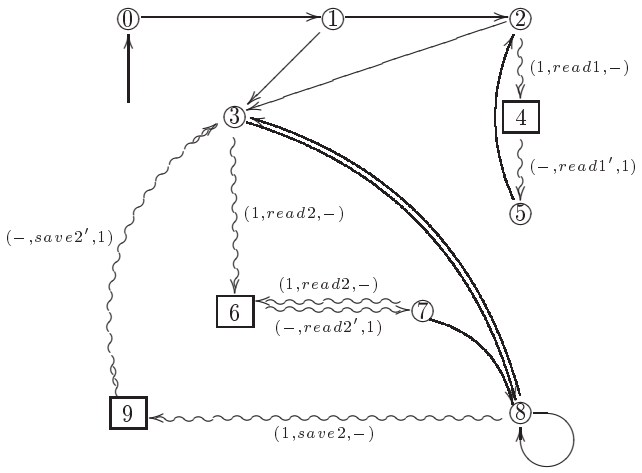


Figure 6: Sets $N_{\mathcal{D}B,\mathcal{L}_{\mathcal{C}_0}}$, $\mathcal{L}^{N\rightarrow}_{\mathcal{D}B,\mathcal{L}_{\mathcal{C}_0}}$ and $\mathcal{L}^{N\leftarrow}_{\mathcal{D}B,\mathcal{L}_{\mathcal{C}_0}}$.

Notation. *Let* $\mathcal{D}$ *be a dynamic system model and* $X \subseteq \mathcal{L}_{\mathcal{C}_0}$. *Then* $\|\mathcal{D}\|_X$ *denotes the minimal* $i \in \mathbb{N}_0$ *such that for each* $j \in \mathbb{N}_0$ *and each reachable state* q *of the automaton* $\mathcal{D}_j$, *the number of clients that are in the state* q *in a cycle of service* X *is less or equal to* i, *if there is any. If there is no such* j *then* $\|\mathcal{D}\|_X = \infty$.

Example 6.1. In the model $\mathcal{D}B$ from Figure 2 at any time, the clients can access either reading or writing on the database. If the database serves the reading, there can be at most two clients that are using it, and if the writing is served, there can be at most one client using the service. Hence it holds that:

- $\|\mathcal{D}B\|_{\{(1,read1,-),(-,read1',1),(1,read2,-),(-,read2',1)\}} = 2$,
- $\|\mathcal{D}B\|_{\{(1,save2,-),(-,save2',1)\}} = 1$ and
- $\|\mathcal{D}B\|_{\mathcal{L}_{\mathcal{C}_0}} = 2$.

Notation. *Let* $\mathcal{D}$ *be a dynamic system model,* $X \subseteq \mathcal{L}_{\mathcal{C}_0}$. *As the sets of states of automata* $\{\mathcal{C}_i\}_{i\in\mathbb{N}}$ *are identical, we denote:*

- $N^{\rightarrow}_{\mathcal{D},X} = \{q \in N_{\mathcal{D},X} \mid \exists q' \notin N_{\mathcal{D},X} \colon (q,l,q') \in \delta_1\}$ *– the states of the automaton* $\mathcal{C}_1$ *that are not in a cycle of service, and from which there is a transition to a state in a cycle of service* X.
- $N^{\leftarrow}_{\mathcal{D},X} = \{q \in N_{\mathcal{D},X} \mid \exists q' \notin N_{\mathcal{D},X} \colon (q',l,q) \in \delta_1\}$ *– the states of the automaton* $\mathcal{C}_1$ *that are not in a cycle of service, and to which there is a transition from a state in a cycle of service* X.
- $\mathcal{L}^{N\rightarrow}_{\mathcal{D},X}$ *– labels, over which a transition from a state in* $N_{\mathcal{D},X}$ *to a state in* $Q \setminus N_{\mathcal{D},X}$ *exists in the automaton* $\mathcal{C}_1$.
- $\mathcal{L}^{N\leftarrow}_{\mathcal{D},X}$ *– labels, over which a transition from a state in* $Q \setminus N_{\mathcal{D},X}$ *to a state in* $N_{\mathcal{D},X}$ *exists in the automaton* $\mathcal{C}_1$.

Example 6.2. For the dynamic system model $\mathcal{D}B$ from Figure 2, it holds that

$\|\mathcal{D}B\|_{\mathcal{L}_{\mathcal{C}_0}} = 2$,

$N_{\mathcal{D}B,\mathcal{L}_{\mathcal{C}_0}} = \{0,1,2,3,5,7,8\}$,
$N^{\rightarrow}_{\mathcal{D}B,\mathcal{L}_{\mathcal{C}_0}} = \{2,3,7,8\}$,
$N^{\leftarrow}_{\mathcal{D}B,\mathcal{L}_{\mathcal{C}_0}} = \{3,5,7\}$,

$\mathcal{L}^{N\rightarrow}_{\mathcal{D}B,\mathcal{L}_{\mathcal{C}_0}} = \{(1,read1,-),(1,read2,-),(1,save2,-)\}$,
$\mathcal{L}^{N\leftarrow}_{\mathcal{D}B,\mathcal{L}_{\mathcal{C}_0}} = \{(-,read1',1),(-,read2',1),(-,save2',1)\}$.

Figure 6 illustrates these terms graphically – the states from

$N_{\mathcal{DB},\mathcal{L}_{\mathcal{C}_0}}$ are depicted as ⓘ, the transitions with labels from $\mathcal{L}^{N\rightarrow}_{\mathcal{DB},\mathcal{L}_{\mathcal{C}_0}}$ and $\mathcal{L}^{N\leftarrow}_{\mathcal{DB},\mathcal{L}_{\mathcal{C}_0}}$ are depicted as $\rightsquigarrow$.

As the value $\|D\|_X$ is algorithmically computable, the following lemma helps in finding the under-approximation of $|\mathcal{D}|_X$ and it implies that if $\|\mathcal{D}\|_X = \infty$, then also $|\mathcal{D}|_X = \infty$. In the next sub-section, we more demonstrate possible usage of the value $\|\mathcal{D}\|_X$ for computing the over-approximation of $|\mathcal{D}|_X$.

Lemma 6.1. *Let $\mathcal{D}$ be a dynamic system model and $X \subseteq \mathcal{L}_{\mathcal{C}_0}$. If $\mathcal{L}^{N\rightarrow}_{\mathcal{D},X} \cap \mathcal{L}^{N\leftarrow}_{\mathcal{D},X} = \emptyset$ then $\|\mathcal{D}\|_X \leq |\mathcal{D}|_X$.*

Proof. See appendix.

Example 6.3. The previous lemma together with Example 6.1 for the model $\mathcal{DB}$ (Figure 2) claims that
- $|\mathcal{DB}|_{\{(1,read1,-),(-,read1',1),(1,read2,-),(-,read2',1)\}} \geq 2$,
- $|\mathcal{DB}|_{\{(1,save2,-),(-,save2',1)\}} \geq 1$ and
- $|\mathcal{DB}|_{\mathcal{L}_{\mathcal{C}_0}} \geq 2$.

6.2 Over-approximation of $|\mathcal{D}|_{\mathcal{L}_{\mathcal{C}_0}}$

Here we focus on finding an over-approximation of $|\mathcal{D}|_X$ using the value $\|\mathcal{D}\|_X$. This task is very complicated and for space reasons, we are not going to present here a general algorithm for computing this value. On the other hand, the computation of the over-approximation is relatively straightforward in many specific cases, and if X is very small (two to four items), it usually suffices to employ the observation that:

"If for a dynamic system $\mathcal{D}$ the automaton $\mathcal{D}_n$ generates all possible runs with respect to the observable labels X, then for every $i \in \mathbb{N}$ the automaton $\mathcal{D}_{n+i}$ generates again the same runs. Hence it holds that $|\mathcal{D}|_X \leq n$."

This implies the next observation that the most difficult case of getting the over-approximation of $|\mathcal{D}|_X$ is for $X = \mathcal{L}_{\mathcal{C}_0}$ (when all labels are observable). Hence we concentrate on this case only. And again for space reasons, not on the general algorithm for computing an over-approximation of $|\mathcal{D}|_{\mathcal{L}_{\mathcal{C}_0}}$, but on an algorithm that can be employed for most practically used dynamic systems. More, this algorithm presents the core idea that can drive construction of the general algorithm.

From now on, we suppose that
- every dynamic system $\mathcal{D}$ fulfils $\mathcal{L}^{N\rightarrow}_{\mathcal{D},X} \cap \mathcal{L}^{N\leftarrow}_{\mathcal{D},X} = \emptyset$,
- in the model, any transition that can synchronise with the provider has at least one of its states in a cycle of service.

These conditions are very weak and common models (which model requests and responses separately) most likely satisfy it.

Notation. *Let $\mathcal{D}$ be a dynamic system model, X a set of labels and $q \in N_{\mathcal{D},X}$. Then* a set of paths of service starting in *q is the set $ServicePath_{\mathcal{D},X}(q)$ of paths of the automaton $\mathcal{C}_1$ that start in the state q, finish in a state $q' \in N^{\leftarrow}_{\mathcal{D},X}$, and none of their internal states is in $N_{\mathcal{D},X}$.*
For $\pi \in ServicePath_{\mathcal{D},X}(q)$ we denote $\pi_{observable}$ the sequence that results from π after removing the states and labels that can not be used for synchronisation with the provider and $ServiceOPath_{\mathcal{D},X}(q)$ denotes the set $\{\pi_{observable} \mid \pi \in ServicePath_{\mathcal{D},X}(q)\}$.

Example 6.4. For the dynamic system model $\mathcal{DB}$ from Figure 2 the following holds:

$N^{\rightarrow}_{\mathcal{DB},\mathcal{L}_{\mathcal{C}_0}} = \{2,3,7,8\}$

$ServicePath_{\mathcal{DB},\mathcal{L}_{\mathcal{C}_0}}(2) = \{2,{}_{(1,read1,-)},4,{}_{(-,read1',1)},5\}$,
$ServicePath_{\mathcal{DB},\mathcal{L}_{\mathcal{C}_0}}(3) = \{3,{}_{(1,read2,-)},6,{}_{(-,read2',1)},7\}$,
$ServicePath_{\mathcal{DB},\mathcal{L}_{\mathcal{C}_0}}(7) = \{7,{}_{(1,read2,-)},6,{}_{(-,read2',1)},7\}$,
$ServicePath_{\mathcal{DB},\mathcal{L}_{\mathcal{C}_0}}(8) = \{8,{}_{(1,save2,-)},9,{}_{(-,save2',1)},3\}$.

$ServiceOPath_{\mathcal{DB},\mathcal{L}_{\mathcal{C}_0}}(2) = \{{}_{(1,read1,-),(-,read1',1)}\}$,
$ServiceOPath_{\mathcal{DB},\mathcal{L}_{\mathcal{C}_0}}(3) = ServiceOPath_{\mathcal{DB},\mathcal{L}_{\mathcal{C}_0}}(7) = \{{}_{(1,read2,-),(-,read2',1)}\}$,
$ServiceOPath_{\mathcal{DB},\mathcal{L}_{\mathcal{C}_0}}(8) = \{{}_{(1,save2,-),(-,save2',1)}\}$.

Notation. *For a dynamic system model $\mathcal{D}$ and the set X, the automaton $\mathcal{C}_i$ restricted to the states that are in $N_{\mathcal{D},X}$ is denoted $\overline{\mathcal{C}_i}$.*

Example 6.5. For the dynamic system model $\mathcal{DB}$ from Figure 2 and $X = \mathcal{L}_{\mathcal{C}_0}$, the automaton $\overline{\mathcal{C}_1}$ contains all states depicted in Figure 6 but it contains only the transitions which are depicted as $\longrightarrow$.

Notation. *The subset of $N^{\rightarrow}_{\mathcal{D},X}$ reachable from the initial state of $\overline{\mathcal{C}_1}$ is denoted $Ninit^{\rightarrow}_{\mathcal{D},X}$.*

Example 6.6. For $\mathcal{DB}$ as the dynamic system model from the Figure 2, $Ninit^{\rightarrow}_{\mathcal{DB},\mathcal{L}_{\mathcal{C}_0}} = \{2,3\}$. Figure 6 shows, that those states are the only states in $N^{\rightarrow}_{\mathcal{DB},\mathcal{L}_{\mathcal{C}_0}}$ reachable from the initial state only via transitions denoted $\longrightarrow$.

Definition 6.2. Let $\mathcal{D}$ be a dynamic system model and X be a set and $q \notin N^{\rightarrow}_{\mathcal{D},X}$. The set of maximal paths in a cycle of service starting in q, denoted $SCPath_{\mathcal{D},X}(q)$, is the set of all maximal paths from q which do not contain a state from $N^{\rightarrow}_{\mathcal{D},X}$ and which do not contain any state twice.
Let $\pi \in SCPath_{\mathcal{D},X}(q)$ for some $q \notin N^{\rightarrow}_{\mathcal{D},X}$, then π_{trace} corresponds to the path π, from which there have been removed all the labels and states that either belong to the set $N_{\mathcal{D},X}$ or do not belong to $N_{\mathcal{D},X}$ and are succeeded by a state not belonging to $N_{\mathcal{D},X}$. Thus π_{trace} is a sequence of states not belonging to $N_{\mathcal{D},X}$ whose successor in π belongs to $N_{\mathcal{D},X}$.
For $q \in Q$, let $MaxSCTrace_{\mathcal{D},X}(q)$ denotes a subset of the set $\{\pi_{trace} \mid \pi \in SCPath_{\mathcal{D},X}(q)\}$, which contains only the maximal sequences.
Let $MaxCSTrace_{\mathcal{D},X} = \bigcup_{q \in Ninit^{\rightarrow}_{\mathcal{D},X}} MaxSCTrace_{\mathcal{D},X}(q)$.

Example 6.7. For the dynamic system model $\mathcal{DB}$ depicted in Figure 2 it holds:
$\{\pi_{trace} \mid \pi \in SCPath_{\mathcal{DB},\mathcal{L}_{\mathcal{C}_0}}(2)\} = \{4\}$,
$\{\pi_{trace} \mid \pi \in SCPath_{\mathcal{DB},\mathcal{L}_{\mathcal{C}_0}}(3)\} = \{6;\ 6,9\}$,
$MaxSCTrace_{\mathcal{DB},\mathcal{L}_{\mathcal{C}_0}}(2) = \{4\}$,
$MaxSCTrace_{\mathcal{DB},\mathcal{L}_{\mathcal{C}_0}}(3) = \{6,9\}$,
$MaxSCTrace_{\mathcal{DB},\mathcal{L}_{\mathcal{C}_0}} = \{4;\ 6,9\}$.

The following lemma contains two preconditions. The first one requires that anytime after serving a client, the client must be able to launch the same serving again. This precondition is requisite. However if it is not fulfilled, $|\mathcal{D}| = \infty$ very likely occurs. The second precondition simplifies the notation and it could be weakened if necessary.

Lemma 6.2. *Let $\mathcal{D}$ be a dynamic system model such that:*
- $\forall q \in N^{\rightarrow}_{\mathcal{D},\mathcal{L}_{\mathcal{C}_0}}$, $\pi = {}_{q_0,l_0,\ldots,l_n,q_{n+1}} \in ServicePath_{\mathcal{D},\mathcal{L}_{\mathcal{C}_0}}(q)$

$\exists q'_0 \in N^{\rightarrow}_{\mathcal{D},\mathcal{L}_{\mathcal{C}_0}}$ *reachable in* $\overline{\mathcal{C}_1}$ *from the state* q_{n+1} *and* $\pi' \in ServicePath_{\mathcal{D}}(q'_0) : \pi_{observable} = \pi'_{observable}$

• $\forall q \in N_{\mathcal{D},\mathcal{L}_{\mathcal{C}_0}}$, $\pi_1, \pi_2 \in ServicePath_{\mathcal{D}}(q)$*: last state of* π_1 *is equal to last state of* π_2*.*

Then

$$|\mathcal{D}|_{\mathcal{L}_{\mathcal{C}_0}} \leq 1 + \|\mathcal{D}\|_{\mathcal{L}_{\mathcal{C}_0}} \cdot \sum_{\pi \in MaxSCTrace_{\mathcal{D},\mathcal{L}_{\mathcal{C}_0}}} |\pi|,$$

where $|\pi|$ *denotes the number of states in* π*.*

Proof. See appendix.

Example 6.8. For the dynamic system model $|\mathcal{D}B|_{\mathcal{L}_{\mathcal{C}_0}}$ from Figure 2 it holds $\|\mathcal{D}B\|_{\mathcal{L}_{\mathcal{C}_0}} = 2$, $MaxTrace_{\mathcal{D}B} = \{4;\ 6, 9\}$ (see Example 6.3, 6.7) and this system fulfils preconditions of Lemma 6.2. Thus Lemma 6.1 and Lemma 6.2 claim

$$2 \leq |\mathcal{D}B|_{\mathcal{L}_{\mathcal{C}_0}} \leq 1 + 2 \cdot (1 + 2) = 7.$$

Therefore for the dynamic system model $|\mathcal{D}B^{impl}|_{\mathcal{L}_{\mathcal{C}_0}}$ from Example 5.3 it holds

$$|\mathcal{D}B^{impl}|_{\mathcal{L}_{\mathcal{C}_0}} \leq 7.$$

7. CONCLUSIONS AND FUTURE WORK

The paper introduces our approach to the verification of dynamic systems. It is based on getting a value n, which guarantees that if the examined property is not violated on the system with less then n dynamic components, it will always hold on the system, no matter how many dynamic components are going to be used during its execution. This result is possible thanks to the assumption, that dynamic components share a common type, which makes their behaviour predictable. Besides the proofs of the propositions constituting this result, we also present the intuition for designing algorithms for getting the approximation of n.

Our technique is applicable not only to the verification of dynamic systems. It can be very helpful also during modelling of a dynamic system, because it can be used to check that the model does not have any unpredictable behaviour comparing to the real system (see [12] for discussion of this). In this case, also verification of simple properties, or information about the number of clients that can be served by a provider, can be interesting.

In future, we aim to finish implementation of the algorithms and extend them also to more general types of dynamic models. We also aim at broadening the set of properties and evaluating the approach thoroughly on realistic case studies.

8. REFERENCES

[1] J. Adámek. Addressing Unbounded Parallelism in Verification of Software Components. In *SNPD*, pages 49–56, 2006.

[2] T. Ball and S. K. Rajamani. Automatically Validating Temporal Safety Properties of Interfaces. *Lecture Notes in Computer Science*, 2057:103+, 2001.

[3] J. Barnat, L. Brim, I. Černá, P. Moravec, P. Ročkai, and P. Šimeček. Divine - A Tool for Distributed Verification. In *Proc. of the 18th International Conference CAV'06*, volume 4144 of *LNCS*, pages 278–281. Springer, 2006.

[4] T. Barros, L. Henrio, and E. Madelaine. Behavioural models for hierarchical components. In *Proceedings of the SPIN 2005 Workshop*, pages 154–180, San Francisco, USA, August 2005. LNCS Springer-Verlag.

[5] L. Brim, I. Černá, P. Vařeková, and B. Zimmerova. Component-Interaction Automata as a Verification-Oriented Component-Based System Specification. In *Proceedings of SAVCBS'05*, pages 31–38, Lisbon, Portugal, September 2005.

[6] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, USA, January 2000.

[7] J. Corbett, M. Dwyer, and J. Hatcliff. Expressing checkable properties of dynamic systems: The Bandera Specification language, 2000.

[8] F. Plasil and S. Visnovsky. Behavior protocols for software components. *IEEE Transactions on Software Engineering*, 28(11):1056–1076, November 2002.

[9] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science*, pages 46–57. IEEE Computer Society Press, 1977.

[10] A. Rensink, Á. Schmidt, and D. Varró. Model Checking Graph Transformations: A Comparison of Two Approaches. In H. Ehrig, G. Engels, F. Parisi-Presicce, and G. Rozenberg, editors, *International Conference on Graph Transformations (ICGT)*, volume 3256 of *Lecture Notes in Computer Science*, pages 226–241. Springer-Verlag, 2004.

[11] I. Černá, P. Vařeková, and B. Zimmerova. Component-Interaction Automata Modelling Language. Technical Report FIMU-RS-2006-08, Masaryk University, Faculty of Informatics, Brno, Czech Republic, October 2006.

[12] B. Zimmerova, P. Vařeková, N. Beneš, I. Černá, L. Brim, and J. Sochor. *The Common Component Modeling Example: Comparing Software Component Models*, chapter Component-Interaction Automata Approach (CoIn). To appear in LNCS, 2007.

APPENDIX

A. DEFINITIONS

In the following, you may find the formal definition of the *complete transition space* $\Delta_{\mathcal{S}}$ defined informally in Definition 3.3.

Notation. *Let $\mathcal{I} = \{i_1, i_2, \ldots, i_n\}$ be a nonempty set of integers with $i_1 < \cdots < i_n$, and let Q_i be a set for each $i \in \mathcal{I}$. Then $\Pi_{i\in\mathcal{I}} Q_i$ denotes the set*

$$\{(q_{i_1}, q_{i_2}, \ldots, q_{i_n}) \mid \forall j \in \{1, \ldots, n\} : q_{i_j} \in Q_{i_j}\}.$$

For any $j \in \mathcal{I}$, pr_j denotes a function $pr_j : \Pi_{i\in\mathcal{I}} Q_i \to Q_j$ such that $pr_j((q_i)_{i\in\mathcal{I}}) = q_j$.

Definition A.1. Let $\mathcal{S} = \{(Q_i, Act_i, \delta_i, I_i, H_i)\}_{i\in\mathcal{I}, \mathcal{I}\subseteq\mathbb{N}}$ be a finite composable set of component-interaction automata. Then the *complete transition space* for $\mathcal{S}$ is $\Delta_{\mathcal{S}} = \Delta_{\mathcal{S},old} \cup \Delta_{\mathcal{S},new}$, where

$$\Delta_{\mathcal{S},old} = \big\{(q,(o,a,o'),q') \mid q,q' \in \Pi_{i\in\mathcal{I}} Q_i,\ o,o' \in \mathbb{N}\cup\mathbb{A}\cup\{-\},\ \exists j \in \mathcal{I}:\ ((pr_j(q),(o,a,o'),pr_j(q')) \in \delta_j \wedge \forall i \in (\mathcal{I}\setminus\{j\})\ pr_i(q) = pr_i(q'))\big\}$$

$$\Delta_{\mathcal{S},new} = \big\{(q,(o,a,o'),q') \mid q,q' \in \Pi_{i\in\mathcal{I}} Q_i,\ o,o' \in \mathbb{N}\cup\mathbb{A},\ \exists j,j' \in \mathcal{I}, j\neq j':\ ((pr_j(q),(o,a,-),pr_j(q')) \in \delta_j \wedge (pr_{j'}(q),(-,a,o'),pr_{j'}(q')) \in \delta_{j'} \wedge \forall i \in (\mathcal{I}\setminus\{j,j'\})\ pr_i(q) = pr_i(q'))\big\}$$

B. PROOFS

This appendix provides the reader with the proofs for the lemmas presented in the text.

Proof. Lemma 5.2 Because $\otimes^{\mathcal{F}'}\{\mathcal{C}_0, \mathcal{C}'\}$ is defined and $S_{H'} \subseteq \mathbb{A}$, it can be simply proved that $\mathcal{D}'$ is a dynamic system model by a checking of conditions in the definition of dynamic system model.

To proof that 1), 2) and 3) implies $|\mathcal{D}'|_X \leq |\mathcal{D}|_X$ it suffice to show that

$$\{\sigma_{\Delta,X} \mid \sigma \in Path^{Init}_{Inf}(\mathcal{D}'_k)\} = \{\sigma_{\Delta,X} \mid \sigma \in Path^{Init}_{Inf}(\mathcal{D}'_{|\mathcal{D}|_X})\}.$$

We prove that for each $k \geq |\mathcal{D}|_X$ it is fulfilled

$$\{\sigma_{\Delta,X} \mid \sigma \in Path^{Init}_{Inf}(\mathcal{D}'_k)\} \subseteq \{\sigma_{\Delta,X} \mid \sigma \in Path^{Init}_{Inf}(\mathcal{D}'_{|\mathcal{D}|_X})\},$$

the inverse follows immediately from precondition 3). For $k > |\mathcal{D}|_X$ let $\pi'^k = {}_{((q'^0_0,q'^0_0),q'^1_0,\ldots,q'^k_0),\, l_0,\, ((q'^0_1,q'^0_1),q'^1_1,\ldots,q'^k_1),\, l_1,\, \ldots}$ be a sequence from $\{\sigma_{\Delta,X} \mid \sigma \in Path^{Init}_{Inf}(\mathcal{D}'_k)\}$. Then $\overline{\pi}'^k$ denotes a sequence created from π'^k by

a) skipping actions involving only the automaton $\mathcal{C}'$,
b) replacing internal actions over labels in $Comm(X, S_{H'})$ by external actions of the automaton $\mathcal{C}$ which model the part of communication involving the automaton $\mathcal{C}$,
c) skipping of parts of the states which model states of the automaton $\mathcal{C}'$.

Formally $\overline{\pi}'^k = {}_{f((q'^0_0,q'^0_0,\ldots,q'^k_0),l_0),f((q'^0_1,q'^0_1,\ldots,q'^k_1),l_1),\ldots}$ where the function f is defined:

$$f(q,(o,a,o')) = \begin{cases} \epsilon, & o \in S_{H'},\ o' \in S_{H'}, \\ q,(o,a,-), & (o,a,o') \in Comm(X,S_{H'}),\ o' \in S_{H'}, \\ q,(-,a,o'), & (o,a,o') \in Comm(X,S_{H'}),\ o \in S_{H'}, \\ q,(o,a,o'), & otherwise. \end{cases}$$

From the precondition (3) it follows that there has to exist a path $\pi^k \in \{\sigma_{\Delta,X} \mid \sigma \in Path^{Init}_{Inf}(\mathcal{D}_k)\}$ satisfying either $\overline{\pi}'^k = \pi^k$ or $\overline{\pi}'^k$ is a prefix of π^k. Because $k \geq |\mathcal{D}|_X$, Definition 5.2 claims that a path $\pi^{|\mathcal{D}|_X} \in Path^{Init}_{Inf}(\mathcal{D}_{|\mathcal{D}|_X})$ such that $\pi^k_{\Delta,X} = \pi^{|\mathcal{D}|_X}_{\Delta,X}$ must exist. Preconditions 1) and 2) imply that it is possible to construct from the path $\pi^{|\mathcal{D}|}$ (inversely to the steps a), b) and c)) a sequence $\pi'^{|\mathcal{D}|_X} \in \{\sigma_{\Delta,X} \mid \sigma \in Path^{Init}_{Inf}(\mathcal{D}_{|\mathcal{D}|_X})\}$ such that $\pi'^k_{\Delta,X} = \pi'^{|\mathcal{D}|_X}_{\Delta,X}$. □

Proof. Lemma 5.3 By induction to structure of the formula.

- $\varphi = \mathcal{E}(l)$: Follows directly from Definition 5.4.
- $\varphi = \varphi_1 \wedge \varphi_2$: According to IH, φ_1 and φ_2 are invariant under stuttering. Let ρ and σ be stuttering equivalent paths wrt. $\mathcal{L}_\varphi$. Then $\rho \models \varphi \Leftrightarrow \rho \models \varphi_1 \wedge \rho \models \varphi_2 \Leftrightarrow \sigma \models \varphi_1 \wedge \sigma \models \varphi_2 \Leftrightarrow \sigma \models \varphi$.
- $\varphi = \neg\varphi'$: According to IH, φ is invariant under stuttering. Let ρ and σ be stuttering equivalent paths wrt. $\mathcal{L}_\varphi$. Then $\rho \models \varphi \Leftrightarrow \rho \not\models \varphi' \Leftrightarrow \sigma \not\models \varphi' \Leftrightarrow \sigma \models \varphi$.
- $\varphi = (\varphi_1 \wedge \neg\mathcal{P}(l_1))\ \mathcal{U}\ (\mathcal{P}(l_2) \wedge \varphi_2)$: This covers all three cases enumerated in Lemma 5.3. According to IH, φ_1 and φ_2 are invariant under stuttering. Let ρ and σ be stuttering equivalent paths wrt. $\mathcal{L}_\varphi$. Then $\rho \models \varphi \Leftrightarrow (1) : \exists m \in \mathbb{N}_0 : \rho^m \models \mathcal{P}(l_2) \wedge \rho^m \models \varphi_2$ and $(2) : (\forall n < m : \rho^n \models \varphi_1 \wedge \rho^n \models \neg\mathcal{P}(l_1))$. According to stuttering equivalence of ρ and σ and stuttering invariance of φ_1 and φ_2, the condition (1) is equivalent to the condition $(1') : \exists m' \in \mathbb{N}_0 : \sigma^{m'} \models \mathcal{P}(l_2) \wedge \sigma^{m'} \models \varphi_2$. Moreover, the condition (2) is equivalent to the condition $(2') : \forall n' < m' : \sigma^{n'} \models \varphi_1 \wedge \sigma^{n'} \models \neg\mathcal{P}(l_1))$ (this follows from the third requirement of Definition 5.4). Putting those two equivalencies together, we conclude that $\rho \models \varphi \Leftrightarrow (1) \wedge (2) \Leftrightarrow (1') \wedge (2') \Leftrightarrow \sigma \models \varphi$.
- $\varphi = \varphi_1\, \mathcal{U}\, \varphi_2$: According to IH, φ_1 and φ_2 are invariant under stuttering. Let ρ and σ be stuttering equivalent paths wrt. $\mathcal{L}_\varphi$. Then $\rho \models \varphi \Leftrightarrow \exists m \in \mathbb{N}_0 : \rho^m \models \varphi_2 \wedge \forall n < m : \rho^n \models \varphi_1$. According to Definition 5.4, let k be the maximal index such that $i_k \leq m$. Then it also holds: $\rho \models \varphi \Leftrightarrow \exists i_k \in \mathbb{N}_0 : \rho^{i_k} \models \varphi_2 \wedge \forall n < i_k : \rho^n \models \varphi_1$. Since ρ^{i_k} and σ^{j_k} are stuttering equivalent, $\rho^{i_k} \models \varphi_2 \Leftrightarrow \sigma^{j_k} \models \varphi_2$. By a similar argument we can prove that $\forall n < i_k : \rho^n \models \varphi_1 \Leftrightarrow \forall n' < j_k : \sigma^{n'} \models \varphi_1$. Putting the results together we obtain that $\rho \models \varphi \Leftrightarrow \exists j_k \in \mathbb{N}_0 : \sigma^{j_k} \models \varphi_2 \wedge \forall n' < j_k : \sigma^{n'} \models \varphi_1 \Leftrightarrow \sigma \models \varphi$. □

Proof. Lemma 5.4 For space reasons let shortcut d means $|\mathcal{D}|_X$. Assume that $\mathcal{D}_{d+j} \not\models \varphi_{d+j}$. Let for any $j \in \mathbb{N}$, $\sigma^{d+j} \in Path^{Init}_{Inf}(\mathcal{D}_{d+j})$ be a path such that $\sigma^{d+j} \not\models \varphi_{d+j}$. Then according to Definition 5.2, there exists a sequence from $Path^{Init}_{Inf}(\mathcal{D}_d)$

$$\sigma^d = {}_{(q^0_0,q^0_0,\ldots,q^d_0),l_0,(q^0_1,q^1_1,\ldots,q^d_1),l_1,\ldots}$$

such that $\sigma^{d+j}_{\Delta,X} = \sigma^d_{\Delta,X}$. A path $\sigma'^{d+j} =$

$${}_{(q^0_0,\ldots,q^d_0,q^{d+1}_0,\ldots,q^{d+j}_0),l_0,(q^0_1,\ldots,q^d_1,q^{d+1}_0,\ldots,q^{d+j}_0),l_1,\ldots,}$$

(where $q^{d+1}_0, \ldots, q^{d+j}_0$ are initial states of automata $\mathcal{C}_{d+1}$, $\ldots$, $\mathcal{C}_{d+j}$) is obviously in the set $Path^{Init}_{Inf}(\mathcal{D}_{d+j})$. From the fact that ${\sigma^{d+j}}_{\Delta,X} = \sigma^d_{\Delta,X}$ it follows that there are two finite or infinite sequences $0 = i^1_0 < i^1_1 < \cdots$ and $0 = i^2_0 < i^2_1 < \cdots$ such that for each index $n \geq 0$:

- for each $l \in \bigcup_{i\in\mathbb{N}_0} \mathcal{L}_{\varphi_i}$ all states of the automaton $\mathcal{C}_0$ $pr_1(\sigma^{d+j}(i^1_n)), \ldots, pr_1(\sigma^{d+j}(i^1_{n+1} - 1))$ and $pr_1(\sigma'^{d+j}(i^2_n)), \ldots, pr_1(\sigma'^{d+j}(i^2_{n+1} - 1))$ satisfy $\models E(l)$ or none of them satisfy it,
- labels $\mathcal{L}(\sigma^d, i^1_n), \ldots, \mathcal{L}(\sigma^d, i^1_{n+1} - 2)$ and $\mathcal{L}(\sigma'^d, i^2_n), \ldots, \mathcal{L}(\sigma'^d, i^2_{n+1} - 2) \notin \bigcup_{i\in\mathbb{N}_0} \mathcal{L}_{\varphi_i}$
- labels ${}_{(o_1,a,o'_1)=\mathcal{L}(\sigma^d,i^1_{n+1}-1)}$ and ${}_{(o_2,a,o'_2)=\mathcal{L}(\sigma'^d,i^2_{n+1}-1)}$ model a communication which involves the provider and ${}_{(f(o_1),a,f(o'_1))=(f(o_2),a,f(o'_2))}$, where

$$f(o) = \begin{cases} o, & o \notin \mathbb{N}, \\ *, & o \in \mathbb{N}. \end{cases}$$

Thus according to Definitions 5.6 and 5.7, $\sigma'^{d+j} \not\models \varphi_{d+j}$. So the second condition of the Definition 5.3 implies $\sigma^d \not\models \varphi_d$ and the implication is proved. □

Proof. Theorem 5.1 For space reasons let shortcut d means $|\mathcal{D}|_X$. Suppose that

$$\mathcal{D}_{d+m+j} \not\models \varphi_{d+m+j},$$

then it is sufficient to prove that there exists a path $\sigma \in Path_{Inf}^{Init}(\mathcal{D}_{d+m+j})$, which does not satisfy φ_{d+m+j} and whose transitions do not involve clients with names larger than $d+m$. The fact $\mathcal{D}_{d+m} \not\models \varphi_{d+m}$ then follows immediately from the second condition of the Definition 5.3.

Firstly we define new dynamic systems $\mathcal{D}'$, $\mathcal{D}''$. In this proof $\mathcal{C}'_1, \mathcal{C}'_2, \ldots, \mathcal{C}'_m$ are CI automata $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_m$ with renamed component names such that the triple

$$(\mathcal{C}_0 \otimes (\otimes\{\mathcal{C}'_i\}_{i\in\{1,\ldots,m\}}), \{\mathcal{C}_i\}_{i\in\mathbb{N}}, \mathcal{F})$$

is a dynamic system model. The injection from the set $\{1,\ldots,m\}$ to the set $\mathbb{A}$ which corresponds to this renaming will be denoted r.

Let $\mathcal{D}'$ be a dynamic system model

$$(\mathcal{C}_0 \otimes (\otimes\{\mathcal{C}'_i\}_{i\in\{1,\ldots,m\}}), \{\mathcal{C}_i\}_{i\in\mathbb{N}}, \overline{\mathcal{F}}),$$

where

$$\overline{\mathcal{F}} = \mathcal{F} \cup \{(a_1, act, o_2) \mid a_1 \in \{r(1),\ldots r(m)\}, (o_1, act, o_2) \in \mathcal{F}, o_1 \in \mathbb{N}, o_2 \notin \mathbb{N}\} \cup \{(o_1, act, a_2) \mid a_2 \in \{r(1),\ldots r(m)\}, (o_1, act, o_2) \in \mathcal{F}, o_1 \notin \mathbb{N}, o_2 \in \mathbb{N}\} \cup \{(a_1, act, a_1) \mid a_1 \in \{r(1),\ldots r(m)\}, (o_1, act, o_1) \in \mathcal{F}, o_1 \in \mathbb{N}\}.$$

Dynamic system models $\mathcal{D}'$ and $\mathcal{D}'' = (\mathcal{C}_0, \{\mathcal{C}_i\}_{i\in\mathbb{N}}, \overline{\mathcal{F}})$, the set of labels X and the set $X' = Comm(X, \{r(i)\}_{i\in\{1,\ldots,m\}})$ satisfy preconditions of Lemma 5.2 thus $|\mathcal{D}'|_{X'} \leq |\mathcal{D}''|_X$. Because $\mathcal{D}_i = \mathcal{D}''_i$ for each $i \in \mathbb{N}_0$ it holds $d = |\mathcal{D}|_X = |\mathcal{D}''|_X$ and thus $|\mathcal{D}'|_{X'} \leq d$.

Let $\mathcal{D}_{d+m+j} \not\models \varphi_{d+m+j}$, then Definition 5.6 and the first condition from Definition 5.3 imply that there exists a sequence

$$\pi = (q_0^0, \ldots, q_0^{d+m+j}), l_0, (q_1^0, \ldots, q_1^{d+m+j}), l_1, \ldots$$

in $Path_{Inf}^{Init}(\mathcal{D}_{d+m+j})$ and a subformula φ of formula φ_{d+m+j} such that $\pi \not\models \varphi$ and $\neg\varphi \Rightarrow \neg\varphi_j$ and formula φ does not involve automata modelling components with numerical names larger than m.

Formulas $\{\varphi_i\}_{i\in\mathbb{N}}$ and φ after renaming all names of components by the function r will be denoted $r(\varphi_i)$ and $r(\varphi)$. $r(\pi)$ will denote a sequence $((q_0^0,\ldots,q_0^m),q_0^{m+1}\ldots,q_0^{d+m+j}),r(l_0),((q_1^0,\ldots,q_1^m),q_1^{m+1}\ldots,q_1^{d+m+j}),r(l_1),\ldots$ in $Path_{Inf}^{Init}(\mathcal{D}'_{d+j})$, where $r_{((o,a,o'))}$ is a label $_{(f(o),a,f(o'))}$ such that $_{f(o)}=\begin{cases} o & o\in\mathbb{A}\cup\{-\}, \\ r(o) & o\in\{1,\ldots,m\}, \\ o-m & otherwise. \end{cases}$

It is clear that $r(\pi) \not\models r(\varphi)$ and $\neg r(\varphi) \Rightarrow \neg r(\varphi_{d+m+j})$ so it holds $\mathcal{D}'_{d+j} \not\models r(\varphi)$ and $\mathcal{D}'_{d+j} \not\models r(\varphi_{d+m+j})$.

The formula $r(\varphi)$ does not involve clients, is invariant under stuttering. Moreover a sequence of formulas $\{\varphi'_i = r(\varphi)\}_{i\in\mathbb{N}}$ is in the set $Property(\mathcal{D}', 0)$ and X' contains all labels necessary for verification of formulas $\{\varphi'_i\}_{i\in\mathbb{N}_0}$ Consequently Lemma 5.2 claims $\mathcal{D}'_d \not\models r(\varphi'_d)$. So it exists a sequence $\sigma'_d \in Path_{Inf}^{Init}(\mathcal{D}'_d)$ such that $\sigma'_d \not\models \varphi'_d = r(\varphi)$.

Then for the path $r^{-1}(\sigma') \in Path_{Inf}^{Init}(\mathcal{D}_{d+m})$, where r^{-1} is defined inversely to the function r holds $r^{-1}(\sigma') \not\models \varphi_{d+m}$ (for the same reasons as in previous proof of Lemma 5.4). □

Proof. Lemma 6.1 Let $j \in \mathbb{N}$ and $\pi^{|\mathcal{D}|_X+j} \in Path_{Inf}^{Init}(\mathcal{D}_{|\mathcal{D}|_X+j})$ be arbitrary but fixed. From the prerequisite it is clear that a sequence $\pi^{|\mathcal{D}|_X} \in Path_{Inf}^{Init}(\mathcal{D}_{|\mathcal{D}|_X})$ satisfying $\pi^{|\mathcal{D}|_X+j}_{\Delta,X} = \pi^{|\mathcal{D}|_X}_{\Delta,X}$ must exist. For each $i \in \mathbb{N}$ transitions in $\mathcal{D}_i$ between the sets $N_{\mathcal{D},X}$ and $Q \setminus N_{\mathcal{D},X}$ model a communication of a client which involves provider who performs an action from X. So from the definition of $\pi^{|\mathcal{D}|_X+j}$ and $\pi^{|\mathcal{D}|_X}$ it follows that there exist two finite or infinite sequences $0 = i_0^1 < i_1^1 < \cdots$ and $0 = i_0^2 < i_1^2 < \cdots$ such that for each index $n \geq 0$:

- labels $\mathcal{L}(\sigma^{|\mathcal{D}|_X}, i_n^1), \ldots, \mathcal{L}(\sigma^{|\mathcal{D}|_X}, i_{n+1}^1 - 2)$ and $\mathcal{L}(\sigma'^{|\mathcal{D}|_X}, i_n^2), \ldots, \mathcal{L}(\sigma'^{|\mathcal{D}|_X}, i_{n+1}^2 - 2)$ do not correspond to a communication over a label in $Comm(X, \mathbb{N})$,
- $_{(o_1,a,o'_1)=\mathcal{L}(\sigma^{|\mathcal{D}|_X},i_{n+1}^1-1)}$ and $_{(o_2,a,o'_2)=\mathcal{L}(\sigma'^{|\mathcal{D}|_X},i_{n+1}^2-1)}$ are labels from $Comm(X, \mathbb{N}) \cup X$ and moreover $_{(f(o_1),a,f(o'_1))=(f(o_2),a,f(o'_2))}$, where $f(o) = o$ for $o \notin \mathbb{N}$ and $f(o) = *$ otherwise.

Since $\mathcal{L}^{N\rightarrow}_{\mathcal{D},X} \cap \mathcal{L}^{N\leftarrow}_{\mathcal{D},X} = \emptyset$ and no more than $|\mathcal{D}|_X$ clients can be in $Q \setminus N_{\mathcal{D},X}$ for any state of the path $\pi^{|\mathcal{D}|_X}$, the same property must hold for the path $\pi^{|\mathcal{D}|_X+j}$. Thus for an arbitrary $j \in \mathbb{N}$ and $\pi^{|\mathcal{D}|_X+j} \in Path_{Inf}^{Init}(\mathcal{D}_{|\mathcal{D}|_X+j})$ the number of components, which are in $Q \setminus N_{\mathcal{D},X}$ during the run, is at most $|\mathcal{D}|_X$. □

Proof. Lemma 6.2 For a $q \notin N_{\mathcal{D}}$ and $End_{\mathcal{D}}(q) = \{q_e\}$, let us define $Succs(q_e)$ ($AllSuccs(q_e)$) as the set of states $q' \in N^{\leftarrow}_{\mathcal{D}}$ ($q' \in Q$, respectively) such that there is a path in the automaton $\mathcal{C}_1$ not containing any state twice, containing a state from $Ninit^{\rightarrow}_{\mathcal{D}}$ as the very first state only, passing q_e and finishing in q'. Moreover, let $rank(q_e)$ be a number of occurrences of the state q_e in $MaxTrace_{\mathcal{D}}$ and $Succs\_rank(q_e) = \sum_{q\in Succs(q_e)} rank(q)$. Now we describe how to simulate a path π (with an arbitrary number of involved components) by a path π' where at most $1 + \|\mathcal{D}\| \cdot \sum_{\pi\in MaxTrace_{\mathcal{D}}} |\pi|$ components are involved. Observe that for any state $q_e \in Ninit_{\mathcal{D}}$, $Succs\_rank(q_e) = |MaxTrace_{\mathcal{D}}(q_e)|$. The idea of the proof follows from the fact that it suffices to deal with at most $Succs\_rank(q_e) \cdot \|\mathcal{D}\|$ components to simulate any behaviour of any number of components which change their local states among states $AllSuccs(q_e)$.

Firstly let us suppose that π contains infinite many executions of a cycle of service. π' is then constructed iteratively according to the sequence of transitions in π. Let $t = q \xrightarrow{l} q'$ be the first not yet simulated transition in π. If t is a part of a cycle of service, it is executed in π' as well. Otherwise let $q_e \in N^{\leftarrow}_{\mathcal{D}}$ be a state such that $AllSuccs(q_e)$ is the minimal set $AllSuccs(\_)$ containing the state q. If less than $Succs\_rank(q_e)$ local states of components are in $AllSuccs(q_e)$, then t can be directly simulated in π' since the component executing t is below the limit of $Succs\_rank(q_e)$ components allowed to simulate a transition among the states from $AllSuccs(q_e)$. Otherwise (at least $Succs\_rank(q_e)$ local states of components are in $AllSuccs(q_e)$), by an argument based on induction we can mimic the transition by a transition of one of $Succs\_rank(q_e)$ components whose local state is within the set $AllSuccs(q_e)$. The correctness of this simulation follows from the fact that the number $Succs\_rank(q_e)$ covers all possible distinguishable paths within the states in $AllSuccs(q_e)$.

If π contains a finite number of executions of any cycle of service, then it suffices to mimic in π' the finite sequence of cycles of service as occurred in π and then to let a component to execute a loop outside any cycle of service forever. To mimic executions of cycles of services in π we can use at most $\|\mathcal{D}\| \cdot \sum_{\pi\in MaxTrace_{\mathcal{D}}} |\pi|$ components (see the previous paragraph) and a never-ending execution of a loop outside any cycle of service can be simulated by one extra component. □

Plan-Directed Architectural Change For Autonomous Systems

Daniel Sykes, William Heaven, Jeff Magee, Jeff Kramer
Department of Computing
Imperial College London
{das05, wjh00, j.magee, j.kramer}@imperial.ac.uk

ABSTRACT

Autonomous systems operate in an unpredictable world, where communication with those people responsible for its software architecture may be infrequent or undesirable. If such a system is to continue reliable operation it must be able to derive and initiate adaptations to new circumstances on its own behalf. Much of the previous work on dynamic reconfigurations supposes that the programmer is able to express the possible adaptations before the system is deployed, or at least is able to add new adaptation strategies after deployment. We consider the challenges in providing an autonomous system with the capability to direct its own adaptation, and describe an initial implementation where change in the software architecture of an autonomous system is enacted as a result of executing a reactive plan.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures

General Terms

Management, Design, Reliability

Keywords

Self-adaptive, self-healing, software architecture, dynamic reconfiguration, autonomous systems

1. INTRODUCTION

If the goal of highly reliable autonomous systems is to be realised, then the software used to control such systems must itself be reliable and highly adaptable. Furthermore it should be able to cope with failures in its components.

In this context, we consider adaptation as a modification—at runtime—of the configuration of the software components which make up the system. However we do not preclude other forms of adaptation, such as changing component parameters, or changes at the language level. Architectural change has the advantage of permitting widespread, if not total, change, while keeping the consistency and safety issues present at lower levels to a minimum. Thus we are concerned with medium to large-scale adaptations. Much previous work has focused on systems where each configuration is a self-contained and often predefined entity, or where *repair strategies* describe how to change between configurations. However, in an autonomous context, it is not feasible to consider every possible scenario beforehand, and in effect pre-program the system to cope with all circumstances which may require an architectural change.

In order to effect this arbitrary change, there must be mechanisms in place to enable the autonomous system to derive a new configuration. This requires some notion of a goal which drives the selection process. This may take the form of a functional goal whereby components are selected on the basis of what operations they perform. Alternatively, the goal may be implicit in constraints on the configuration, which may describe architectural, functional, or performance-related restrictions.

In our initial work in this area, we have developed a system which permits arbitrary dynamic reconfiguration by exploiting the presence of a reactive plan which determines the system's behaviour. Reactive plans are generated with a planning tool from high-level goals given by the user. The behaviour of the system is defined by the set of condition-action rules given in the plan. These rules indicate what components will be required to execute the plan.

In Section 2 we discuss some existing work in the area of dynamic component configuration before giving an example that motivates our approach in Section 3. Our approach is then outlined in more detail in Section 4. The paper concludes with a discussion and mention of interesting future work in Section 5.

2. RELATED WORK

Many previous authors have described approaches which assume adaptation can be specified and analysed before the system is deployed. Unfortunately, this is not always the case with autonomous systems.

Zhang, Cheng *et al.* [15] apply formal techniques to show how the safety of a transition from one steady-state program (which may be thought of as an architecture) to another can be guaranteed. They assume that the adaptive transitions are specified by the designer which requires a worst case of

Sixth International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2007), September 3-4, 2007, Cavtat near Dubrovnik, Croatia.

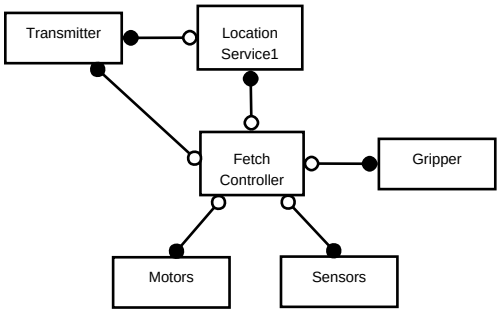


Figure 1: An initial component configuration for the "fetch" task

N^2 transitions for N configurations.

Garlan and Schmerl [5] achieve dynamic change by describing an architectural style [4] for a system and a repair strategy. The repair strategy is a script which modifies the architecture in response to changes in the monitored system properties breaking their associated constraints. Constraints may be on the architecture of the system (as in the usual notion of style) or on the performance of the system. This is a closed-adaptive [10] system since the repairs are specified before deployment. Moreover, this system does not allow architectural change to result from a change in the system's goals.

Dashofy *et al.* [3] use an architectural model and design critics [14] to determine whether a set of changes (an architectural "diff") is safe to apply. They do not directly address when the changes should be applied, but they do allow for an extensible set of repair strategies. Again, these strategies are provided by the user and not derived by the system itself.

Oreizy *et al.* [11] also use an architectural model to ensure changes are valid before they are reflected back into the running implementation. Here descriptions of reconfigurations are provided various parties such as the application vendor.

3. MOTIVATING EXAMPLE

To demonstrate the limitations of current approaches, we consider an example where a mobile autonomous system is deployed and performing a "fetch" operation which requires that it locate, pick up, and return a known object. The software architecture for such a system may resemble that in Figure 1.

The Fetch Controller is responsible for providing operations such as moving to particular locations (while avoiding obstacles), and picking up the object. The Location Service in this case informs the system of its location by communicating with a satellite via the Transmitter.

If at some point during operation, the system's battery no longer has enough power to drive the motors, the system must switch modes in order to use a Beacon component which transmits the system's location in the hope that it will be rescued by another autonomous vehicle, which may have the ability to refuel it. This configuration is shown in

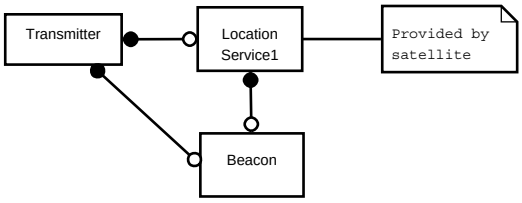


Figure 2: Component configuration following power failure

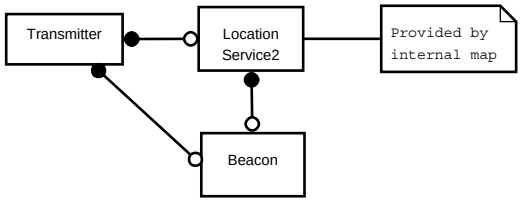


Figure 3: Component configuration following loss of satellite connection

Figure 2.

At this point, the connection to the satellite may be lost (if it moves below the horizon). This prevents the original Location Service (1) from being used, and so the system must find some alternative method for deriving the location. Location Service (2) provides location information based on local information, such as comparing short-range sensor readings to a map of the environment (this may be unreliable). This results in Figure 3. Note in this configuration there is no need for a connection between Location Service (2) and the Transmitter.

One can imagine designing repair strategies for each of these events in isolation, but as the number of possible changes increases it becomes increasingly unlikely that the situation will have been foreseen. Indeed in the worst case $n^m - 1$ repairs must be designed where m is the number of components that can change and n is the number of alternatives.

Hence, we would like to avoid pre-programming repair strategies by having the system derive changes itself.

4. APPROACH

We are experimenting with an approach that derives its own component configurations from *reactive plans* [8]. In the initial planning step, a plan is automatically generated from high-level user goals. This plan describes the behaviour of the system in terms of actions which lead from an initial state to a goal state, without explicit reference to architectural concerns. In particular, there is no correspondence between plan states and configurations. The plan is then submitted to an architecture manager which determines which components are necessary to perform the plan, and instantiates the configuration. The plan interpreter iterates through the rules of the plan to completion, unless a situation is detected which requires reconfiguration or replanning. A brief introduction to reactive plans and their generation is necessary before discussing the derivation of component configurations.

4.1 Generating Reactive Plans

A linear STRIPS-style plan [8] specifies a sequence of actions that are intended to lead from an initial state to a goal state. However, such a plan is not well suited to a non-deterministically changing environment in which a change in the environment may cause an action to lead to a state other than that expected at the time the plan was generated. If this happens, a plan must be regenerated taking into account the changed environment.

A reactive plan, on the other hand, is a plan that accommodates a non-deterministically changing environment by prescribing an action towards a given goal for each state from which that goal is reachable. Execution of such a plan proceeds by determining the current state of the environment, selecting the action prescribed for that state by the plan, performing it and then determining the new state etc. By covering all states from which the goal is reachable, it does not matter if the new state following an action is the "expected" state or not. As long as the goal is reachable from this state, execution of the plan may continue.

In our system, reactive plans are generated using planning-as-model-checking technology [6]. A *domain description* is specified in SMV [13], comprising state predicates and pre- and postcondition constraints on the actions that may be performed. This description is submitted to the Model-Based Planner tool (MBP) [12] along with a specification of the initial state I and a goal G, typically expressed in CTL [2].

The output of MBP is a set of condition-action rules such that each condition corresponds to a state in the environment from which the goal is achievable and each action is an action that may be performed in that state. Formally, this reactive plan is a partial map

$$P : S \to A$$

where S is the set of states in the state space described by the predicates of the domain description and A is the set of actions specified in the domain description. A state $s \in S$ is represented as a set of predicates $\{P_1, P_2, \ldots, P_n\}$.

If a reactive plan P is considered alongside the domain description from which it was generated, it can be represented as a labelled transition system

$$P_{LTS} = \{I, S_P, S_G, T\}$$

where I is the initial state submitted to the planning tool, S_P is the domain of P, $S_G \subseteq S_P$ is the set of states that satisfy G, and $T \subseteq S_P \times A \times S_P$ is a transition relation with transitions labelled with actions in A. T is constructed from P and the domain description so that for all states s in the domain of P there is a state s' such that $(s, a, s') \in T$ if and only if $a = P(s)$. In other words, the transition relation simply picks up the information about what state an action may lead to—which is missing from the information provided by a reactive plan alone—from the postcondition specifications of actions in the domain description.

As a small example, Figure 4 shows a reactive plan for the given domain description. LTS A represents a domain description with start state in the top left corner and goal state

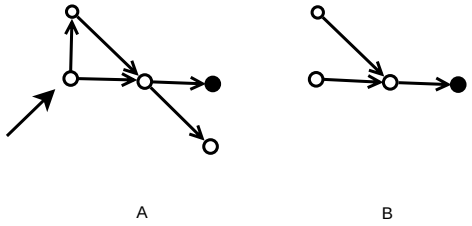


Figure 4: Example plan (B) generated from a domain description (A).

```
...
VAR
object1_location : {loc1, loc2};
rover1_location : {loc1, loc2};
rover1_has : {object1, 0};
rover1_no_power : boolean;
...
INIT
object1_location=loc1 &
rover1_location=loc2 &
rover1_no_power=0 &
rover1_has=0
...
action: {
   rover1_replenish_battery,
   rover1_move_to_loc1,
   rover1_move_to_loc2,
   rover1_pickup,
   rover1_drop,
};
...
ASSIGN next(rover1_location) :=
case
(action = rover1_move_to_loc1) : loc1;
(action = rover1_move_to_loc2) : loc2;
1 : rover1_location;
esac;
...
ASSIGN next(rover1_has) :=
case
(action = rover1_pickup)
   & rover1_location=object1_location : object1;
(action = rover1_drop) : 0;
1 : rover1_has;
esac;
...
-- etc
```

Figure 5: Example domain description fragment input to MBP.

```
-- case 1 (satisfies goal)
(case (and (= object1_location loc2))
   (done))
...
-- case i
(case (and
      (= object1_location loc1)
      (= rover1_location loc1)
      (= rover1_has object1)
      (= rover1_no_power 0))
   (action rover1_move_to_loc2))
...
-- case j
(case (and
      (= object1_location loc1)
      (= rover1_location loc1)
      (= rover1_has item1)
      (= rover1_no_power 1)
   (action rover1_replenish_battery))
...
-- case k
(case (and
      (= object1_location loc1)
      (= rover1_location loc1)
      (= rover1_has 0)
      (= object1_no_power 0))
   (action object2_pickup))
...
-- etc
```

Figure 6: Example reactive plan fragment output by MBP.

in black. LTS B represents a reactive plan which includes all states from which the goal is reachable. Where there are multiple paths to the goal, the shortest is selected. Paths which do not lead to the goal are pruned.

An example domain description, as submitted to MBP, is partially shown in Figure 5. The syntax here is that for the SMV model checker (the back end to MBP). However, the relevant elements of this example are on the whole self-explanatory. The section headed *VAR* list the predicates used to describe the state space. For instance, predicates include *object1_location*, which specifies whether *object1* is in *loc1* or *loc2*. It should be noted that the locations in a domain description are symbolic and are mapped to real locations when the system executes. The section headed *INIT* defines an initial state. Next, the domain description lists the performable actions. Actions are specified through SMV *ASSIGN* blocks, which describe the transitions between states that the system can make. Each block take the form of a case statement. To the left of the colon in each case is the precondition (for technical reasons, actions are treated as part of the precondition) and to the right is the corresponding postcondition. For instance, in the first case of the block describing how the predicate *rover1_location* can evolve, the postcondition for the action *rover1_move_to_loc1* is *rover1_location=loc1*.

This domain description is submitted to MBP along with a goal. Consider, for example, the specified objective for a rover *rover1* to fetch an object *object1* from location *loc1* and bring it to *loc2*. This objective can be captured by a goal stating that in some future state the location of *object1*

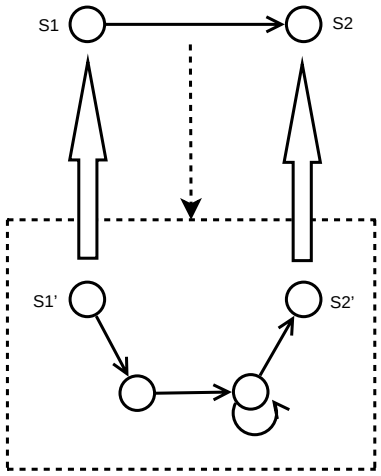


Figure 7: Refinement of an action between states S1 and S2 by a subplan.

is *loc2*. In CTL we capture this as follows:

EF*object1_location=loc2*

(where **EF** may be read "there exists some future state such that .."). Submitting the domain description shown in Figure 5 together with this goal to MBP, we get the plan partially shown in Figure 6. Each case of the plan describes a state from which the goal is reachable and maps that state to an action from the domain description.

As with all model-checking technology, the size of the state space—here determined by the number of predicates in the domain description—becomes a problem in all but the most trivial cases. To address this issue, we organise our domain description into a hierarchy of partial descriptions, generating subplans for each. In this way, each subplan addresses only a part of the overall goal and need only be generated from a partial description of the domain, reducing the number of predicates–and thus size of state space—in each plan generation.

A detailed description of this process is beyond the scope of this paper. However, the core idea is that some of the actions specified in the domain description are "primitive actions" and others are "compound actions". Primitive actions can be performed directly by the system, i.e., it is assumed that they are directly implemented by some component. Compound actions, on the other hand, are abstractions of more complex tasks that require planning. As such, when a plan is being executed and a compound action is encountered, a subplan is generated on the fly for the compound action. The plan is generated with the current state as initial state, postcondition of the compound action as the goal, and a reduced domain description relevant to the performance of the compound action.

Formally, the LTS representing the subplan generated for a compound action is a refinement of the transition representing that action in the original plan. This relationship is depicted in Figure 7, where the transition between states S1 and S2 at the top is refined by the LTS below. The set

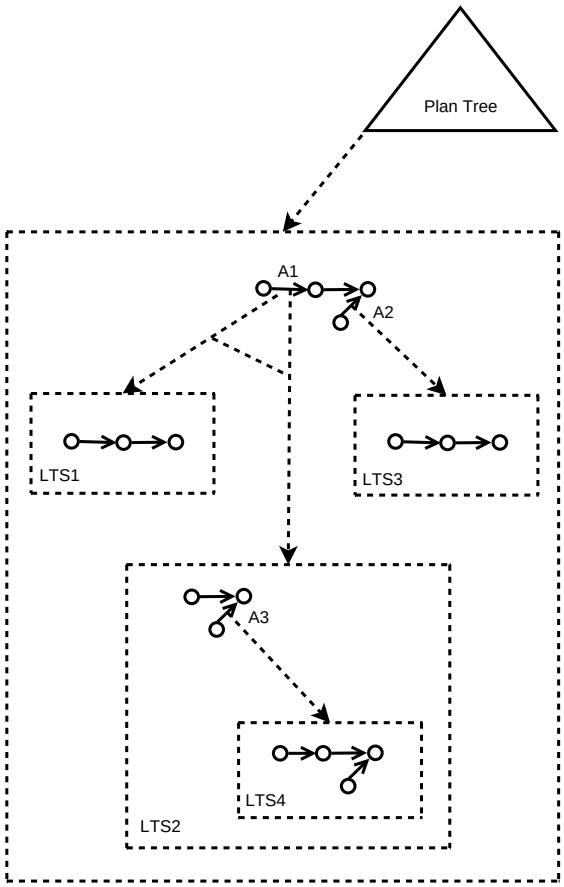


Figure 8: An example plan hierarchy

of predicates describing S1' (resp. S2') implies the set of predicates describing S1 (resp. S2). The dotted arrow and box depict this refinement relationship and will continue to do so in the sequel.

When executing a hierarchical plan of this kind, the system will request the generation of a subplan when a compound action is encountered, execute this subplan, and then jump back to and continue executing the original plan. As such, plan execution can be thought of as resembling depth first traversal of a tree. This is illustrated in Figure 8, which shows an example subtree in the planning hierarchy. It is assumed that the LTS containing actions A1 and A2 is itself a refinement of some transition above it in the plan tree. Here, it can be seen that action A1 has at least two possible refinements, LTS1 and LTS2. Though both are shown here, during execution the planning tool will pick only one alternative at a time and execution will jump to whatever subplan is first chosen. Only if this subplan fails will a request for an alternative be issued by the system. In this case, traversal of the tree would backtrack and execution will jump to LTS2, which in turn contains an action A3 which is refined by LTS4. Again, the dotted lines and boxes depict refinement relationships.

It is possible for execution of a reactive plan to go into a cycle and never reach a goal state. If execution falls into a cycle, we trigger a timeout and have the system request

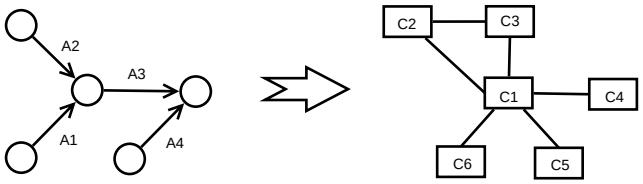


Figure 9: Example component configuration determined by actions of a plan

a different plan. The new plan will typically be generated without the action that caused the cycle, since it is likely that the environment has changed in such a way that the actual effect of this action is no longer accurately modelled by the current domain description.

The new plan will be generated from the point in the hierarchy that the previous plan had been, i.e., unless the plans are at the root of the hierarchy, both new and old plans should be alternative refinements of the same transition in a common parent plan. If no new plan can be generated with the current domain description then the system must backtrack and request a new plan from a node further up the tree.

4.2 Deriving Component Configurations

Since reactive plans are composed of condition-action rules, we are able to use the actions of the plan to derive the functional requirements of the system's architecture. For example, the presence of a move operation in the plan clearly indicates that the configuration must include a component which provides a suitable implementation of this action. We assume that the component responsible for the architectural change (which we call the architecture manager) is aware of the components which provide implementations of actions. For the purposes of deriving component configurations, we do not regard the manager as part of the architecture. Actions may be associated with particular interface types, and the manager selects components which implement the relevant interface. The mapping from actions to interfaces need not be fixed, and could be extended as new components become available.

Given the set of components required for their functionality, the manager can then construct a complete configuration by considering the required interfaces of those components. For example, the component implementing the move operation may require motor and sensor controllers, or a component which provides mapping information. These must also be instantiated and connected to the relevant ports of the action component. In the case where a component is already instantiated, it should be reused. Figure 9 shows a reactive plan and a corresponding architecture. Actions A1 and A3 may be implemented by C1 and actions A2 and A4 may be implemented by C2. The remaining components are found by considering the requirements of C1 and C2.

It may be the case that multiple components provide the same functionality, but have differing non-functional properties. For example, some implementations may require more CPU attention or provide unreliable results. We hope to develop a mechanism whereby the "best" alternative can be

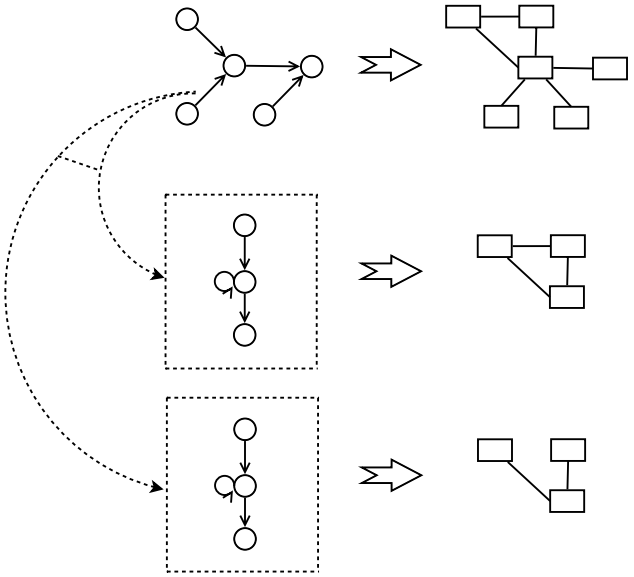


Figure 10: Overview of dynamic changes driven by automatic generation of plans. The two subplans represent alternative refinements of an action in the original LTS. Different component configurations are derived from each alternative.

selected in a given situation.

There is a trade-off to be made in terms of the number of actions a particular component can perform against how often the architecture needs to change. Clearly this depends on particular component implementations and the level of abstraction the plan designer has built into the plans. In our early implementation, components were selected and changed at every step in the plan on the assumption that a component implements only one action. However, this is clearly detrimental to efficiency. Hence, we are moving toward a system whereby a particular plan can be scanned before starting, to construct a configuration that contains components which will implement all the required actions. This is a more natural approach since the architecture is expected to be able to perform the task without changing in the absence of problems.

The exception to this principle is that the architecture may be required to change when a particular abstract action is decomposed into a subplan, since the (concrete) actions contained in the subplan were not known when the parent plan was generated. Furthermore, an abstract action may be decomposed into different subplans in different circumstances. This leads us to the diagram in Figure 10 wherein each plan and subplan has an associated architecture. The top-level plan contains an action which has two possible refinements, resulting in two potential configurations.

We do not employ a verification mechanism for configurations because we regard them as correct in the sense that they must at least provide sufficient functionality to perform the actions of the current plan. Furthermore it is reasonable to assume that the mapping between concrete actions in the plan and their implementations is correct. In other words, the two following constraints are relevant:

$$\forall a \in plan.\exists i \in arch.(a \in i) \wedge \exists c \in arch.prov(c, i)$$

$$\forall c, i \in arch.req(c, i) \longrightarrow \exists c2 \in arch.prov(c2, i)$$

Where $arch$ denotes the set of components and interfaces in the current configuration, $plan$ denotes the current plan (reduced to a set of actions) and $prov(c, i)$ and $req(c, i)$ denote that component c provides (respectively requires) interface i. An interface i is regarded as containing a set of (names of) actions a. The first constraint states that for all actions, there should be a component for the corresponding interface, and the second constraint is simply that all component requirements are satisfied. We do not (at this point) employ further structural, compatibility, or performance constraints.

The mechanisms described so far account for the first configuration change required in the example of Section 3. The Fetch Controller provides the functionality needed for actions in the fetch plan (Figure 6), and the other components in the initial configuration are requirements of the Fetch Controller. The plan checks the *rover1_no_power* predicate which causes a *rover1_replenish_battery* action. When this action is encountered, a subplan is generated which enables a rescue beacon which cycles, transmitting the current location, until the battery is refuelled. As discussed in the previous section, this subplan is a refinement of the *rover1_replenish_battery* action. In this case, the Beacon component is selected because it provides that functionality, and the Location Service (1) and the Transmitter are retained as dependencies of the Beacon, giving Figure 2.

The second case requires the system to cope with entirely unexpected faults. Our approach is to allow the manager to request replanning without using the actions associated with the component that has failed (detected by some suitable mechanism). Of course, planning comes at some cost, so there is a trade-off to be made between that and allowing the manager to perform low-level changes independently, such as substituting a component which implements the same interfaces for the one which failed. It is this latter case which is most appropriate to arrive at Figure 3. The Beacon merely cares about getting location information, and if an alternative implementation is available, it should be used without replanning.

Our implementation of this approach is built upon the Backbone system [9] which allows us to construct arbitrary configurations of components, which are implemented as Java classes. A number of problem domains have been described and executed on a set of Koala robots running a JVM.

5. DISCUSSION AND FUTURE WORK

We have described an initial scheme which addresses the problem of arbitrary dynamic reconfiguration. Reconfiguration is driven by a plan which dictates what functionality the current configuration must provide. Component selection works within the limitations of the current environment which may prevent certain components from being used.

Currently, non-functional and structural constraints on the architecture are not supported. For example, one can imag-

ine a situation where the autonomous system must avoid using components which result in the hardware drawing large amount of power, or where components must be distributed in a particular manner to meet some load balancing constraint.

It remains to be seen whether such constraints can be combined with the reactive plan which at present only prescribes the system's behaviour; the architecture is a consequence of that.

Indeed, another approach would be to employ an explicit architecture plan [1], or include reconfiguration operations within the behavioural plan. One disadvantage of following this path is that the state space for planning becomes larger, with the concomitant reduction in performance.

Other issues we seek to address are those regarding the safety of the adaptation procedure. Clearly if some components are to be replaced, then their dependants must not initiate communications with them for the duration of the change. This is the notion of quiescence [7]. It is especially important for an autonomous system to be able to keep the unaffected parts of the architecture running while reconfiguration is taking place. For the same reasons, components may require special shut down procedures before they are removed from the architecture. For example, any motor control system must ensure those motors are halted before control is released.

6. ACKNOWLEDGEMENTS

The work reported in this paper was funded by the Systems Engineering for Autonomous Systems (SEAS) Defence Technology Centre established by the UK Ministry of Defence.

7. REFERENCES

[1] N. Arshad, D. Heimbigner, and A. L. Wolf. Deployment and dynamic reconfiguration planning for distributed software systems. *Software Quality Journal*, 2003.

[2] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 1999.

[3] E. M. Dashofy, A. van der Hoek, and R. N. Taylor. Towards architecture-based self-healing systems. In *WOSS '02: Proceedings of the first workshop on Self-healing systems*, pages 21–26, New York, NY, USA, 2002. ACM Press.

[4] D. Garlan, R. Allen, and J. Ockerbloom. Exploiting style in architectural design environments. In *SIGSOFT '94: Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering*, pages 175–188, New York, NY, USA, 1994. ACM Press.

[5] D. Garlan and B. Schmerl. Model-based adaptation for self-healing systems. In *WOSS '02: Proceedings of the first workshop on Self-healing systems*, pages 27–32, New York, NY, USA, 2002. ACM Press.

[6] F. Giunchiglia and P. Traverso. Planning as Model Checking. *5th European Conference on Planning*, 1999.

[7] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE Trans. Softw. Eng.*, 16(11):1293–1306, 1990.

[8] Malik Ghallib, Dana Nau, Paolo Traverso. *Automated Planning: Theory and Practice*. Morgan Kaufman, 2005.

[9] A. McVeigh, J. Kramer, and J. Magee. Using resemblance to support component reuse and evolution. In *SAVCBS '06: Proceedings of the 2006 conference on Specification and verification of component-based systems*, pages 49–56, New York, NY, USA, 2006. ACM Press.

[10] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3):54–62, 1999.

[11] P. Oreizy, N. Medvidovic, and R. Taylor. Architecture-based runtime software evolution. In *Proceedings of the 1998 (20th) International Conference on Software Engineering*, pages 177–186, 1998.

[12] P. Bertoli, A. Cimatti, M. Pistore, M. Roveri, P. Traverso. MBP: A Model-Based Planner. *Proc. of IJCAI'01 Workshop on Planning Under Uncertainty and Incomplete Information*, 2001.

[13] P. Bertoli, A. Cimatti, M. Pistore, M. Roveri, P. Traverso. NuSMV 2: An Open Source Tool for Symbolic Model Checking. *Proc. of International Conference on Computer-Aided Verification*, 2002.

[14] J. E. Robbins, D. M. Hilbert, and D. F. Redmiles. Using critics to analyze evolving architectures. In *Joint proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints '96) on SIGSOFT '96 workshops*, pages 90–93, New York, NY, USA, 1996. ACM Press.

[15] J. Zhang and B. Cheng. Modular model checking of dynamically adaptive programs. Technical report, Michigan State University, 2006.

Reachability Analysis for Annotated Code

Mikoláš Janota
School of Computer Science and Informatics
University College Dublin
Belfield, Dublin 4, Ireland
mikolas.janota@ucd.ie

Radu Grigore
School of Computer Science and Informatics
University College Dublin
Belfield, Dublin 4, Ireland

Michał Moskal
Institute of Computer Science
University of Wrocław
ul. Joliot-Curie 15
50-383 Wrocław, Poland
mjm@ii.uni.wroc.pl

ABSTRACT

Well-specified programs enable code reuse and therefore techniques that help programmers to annotate code correctly are valuable. We devised an automated analysis that detects unreachable code in the presence of code annotations. We implemented it as an enhancement of the extended static checker ESC/Java2 where it serves as a check of coherency of specifications and code. In this article we define the notion of semantic unreachability, describe an algorithm for checking it and demonstrate on a case study that it detects a class of errors previously undetected, as well as describe different scenarios of these errors.

Categories and Subject Descriptors

D.2.1 [**Requirements/Specifications**]: Tools; D.2.4 [**Software/Program Verification**]: Formal Methods

General Terms

Verification

Keywords

JML, ESC/Java2

1. INTRODUCTION

Program annotations are logic specifications embedded in the actual program code [16]. They enable programmers to express the intended functionality. Variants of a weakest precondition or a strongest postcondition calculus are used to statically determine whether a program code conforms to its annotations. The extended static checker ESC/Java2 [18] is a tool that attempts to verify annotated Java programs following this approach (Section 2.1).

Empirical evidence shows that automated sanity checking of annotations is desirable [6]. In particular, Leavens et al. [22] propose as one of the challenges for software verification the following: "Provide assistance in specifying libraries of classes." In this article we focus on a particular sanity check — code reachability. Code is unreachable if it is not executed for any possible input. Unreachable code, also known as *dead code*, is often a bug. For example, the Java compiler tries to prevent bugs by disallowing code following a **return** statement.

```
/*@ requires x > 10;
  @ ensures
  @   \result == 1;*/
int withPre(int x) {
 if (x < 10) {
   // not checked
   return 2;
 }
 return 1;
}
```

```
/*@ requires i >= 10;
  @ ensures
  @   \result == i;
  @ ensures
  @   \result < 10;
  @ modifies
  @    \nothing;*/
int libraryFunc(int i);

int useLibraryFunc() {
 int r = libraryFunc(11);
 return 1/0;
}
```

Figure 1: Examples of code that is unreachable once the annotations are taken into account.

Annotations provide extra information about the program, so the notion of code unreachability needs to be extended. Consider the examples in Figure 1. The precondition of the method withPre, expressed by the requires clause, restricts the value of the parameter x to be greater than 10 and the postcondition, expressed by the ensures clause, restricts the return value to be always 1. The **return** statement in the "then" branch of the **if** statement appears to be violating the postcondition. Nevertheless, because of the precondition, this code conforms to its annotation and a static checker like ESC/Java2 will not produce a warning. The fact that a method contains code that is unreachable from the point of the specification is likely to be a bug, either in the specification or in the program code.

The method libraryFunc illustrates a method for which we do not have an implementation (for example because the implementation is proprietary) and we need to rely on its specification. In ESC/Java2 all the methods in the standard Java API are treated in this way. Unfortunately, the specification is inconsistent as it requires the return value to be at least 10 and at the same time to be less than 10. The repercussions of this inconsistency are demonstrated by the useLibraryFunc. The **return** statement in this method seems wrong and yet the extended static checker does not give a warning. The reason for this behavior of the checker

Sixth International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2007), September 3-4, 2007, Cavtat near Dubrovnik, Croatia.

is less obvious than in the previous example and we will explain it in more detail later. Intuitively, as the specification of libraryFunc is inconsistent, from the point of view of the checker the call to that function never terminates and therefore the checker 'believes' that the **return** statement is never executed.

Hence, the problem we address in this article is how to detect unreachable code in the presence of annotations and how we can benefit from such analysis in extended static checking. More specifically, the contributions of the article are as follows: (1) we introduce the notion unreachability for annotated code, (2) we identify several types of unreachable code categorized by their root cause, (3) we present an efficient algorithm for detecting unreachable code, (4) we present an evaluation of the analysis on an existing code base, and (5) an implementation, which is part of ESC/Java2[1].

2. BACKGROUND

Programmers reduce development time dramatically by reusing components that are well documented [20]. In the Java world this is achieved by using `javadoc`, which supports a form of structured documentation [15, 19]. The Java Modeling Language [21] (JML) was designed to allow more formal documentation. Tools can statically check if code and JML-annotations agree. When static checking fails (for example because the code is too complex), the annotations can be compiled into runtime checks. Moreover, unit tests can be generated automatically [5].

The leading static checker for JML-annotated Java is ESC/Java2. Spec$^{\#}$ has a similar architecture and works for annotated C$^{\#}$ programs [2].

2.1 ESC/Java2 Architecture

JML annotations are embedded in Java code as a special form of comments. They are used to specify the behavior of classes and methods in terms of preconditions, postconditions, invariants, and other higher-level constructs. ESC/Java2 checks if code and annotations agree and if there are no runtime exceptions. Methods are checked one at a time, ignoring other methods' implementation and relying on their specification.

For a given JML-annotated method, ESC/Java2 generates a formula, called a *verification condition* (VC), using a strongest postcondition calculus. Further, it tries to prove the verification condition by using an automated theorem prover. If the VC is not proven valid, the checker produces warnings derived from the counterexamples provided by the prover. These warnings describe how the program may violate its JML specification, or in what way the specified program might cause runtime exceptions (such as NullPointerException).

ESC/Java2 performs the translation of JML-annotated Java code to a VC in several stages. This process is schematically depicted in Figure 2.

Given a JML-annotated Java program, the front-end produces an *abstract syntax tree* (AST), which is translated into an intermediate representation called *guarded commands* (GC) [25]; this representation captures both the Java code and its JML annotation. The components that infer invariants [17, 13] work on this representation. Subsequently, loops are translated into structurally simpler commands by a process called *loop desugaring*. ESC/Java2 supports two modes of loop desugaring: One mode is called *loop unrolling* and does not require loop invariants, but it is unsound (see Section 4.3 for more details); the other mode is called *safe desugaring* and treats loops in accord with Hoare logic [16], but requires loop invariants.

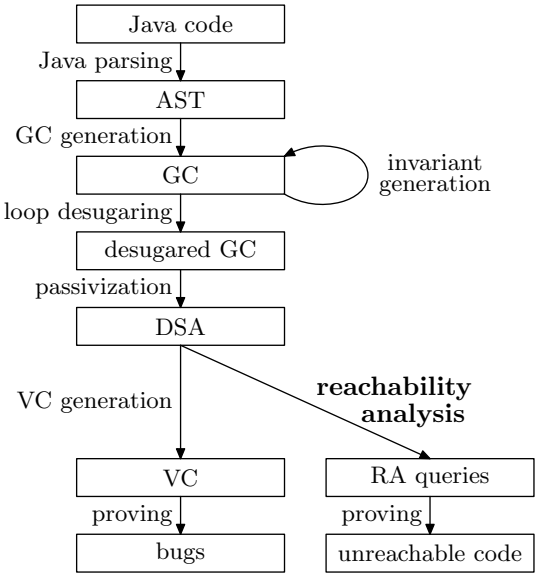


Figure 2: ESC/Java2 architecture

There is an obvious tradeoff between loop unrolling and safe desugaring. The loop unrolling mode may miss some errors as it does not reason about all possible execution traces of the program. The safe loop desugaring does not suffer from this deficiency but it leads to spurious warnings if a strong-enough loop invariant is not provided. Loop invariant generation techniques are used to infer invariants automatically and hence alleviate the annotation burden imposed on the user [10, 13, 23, 17]. Nevertheless, these techniques are computationally expensive and they do not always succeed in finding the proper invariant. In ESC/Java2 loop unrolling is the default loop desugaring mode, since the alternative is not yet practical.

After loop desugaring, the desugared GC is translated into an assignment-free form, or *passive form*, called *dynamic single assignment* (DSA). This is done by ensuring that each variable is assigned-to at most once, which often requires additional variables, and by replacing assignments with assumptions. The main purpose of this particular transformation is to avoid the exponential explosion in the size of the generated VC (see [14] for details).

After the DSA form is generated, the VC is generated from it and sent to a theorem prover. Finally, the output of the prover is processed to provide feedback to the user.

The process described above represents the skeleton of ESC/Java2 and additional analyses can be 'hooked' in this architecture to facilitate the application of the tool. This is the case of the reachability analysis presented in this paper, which is applied on the DSA representation and uses a theorem prover. Since this particular analysis slows down the

[1] `http://kindsoftware.com/products/opensource/ESCJava2/cvs.html`

C	$\mathrm{N}(P, C)$	$\mathrm{W}(P, C)$
skip	P	*false*
assume f	$f \wedge P$	*false*
assert f	$f \wedge P$	$P \wedge \neg f$
$C_1 \,[\!]\, C_2$	$\mathrm{N}(P, C_1) \vee \mathrm{N}(P, C_2)$	$\mathrm{W}(P, C_1) \vee \mathrm{W}(P, C_2)$
$C_1; C_2$	$\mathrm{N}(\mathrm{N}(P, C_1), C_2)$	$\mathrm{W}(P, C_1) \vee \mathrm{W}(\mathrm{N}(P, C_1), C_2)$

Figure 3: Strongest postcondition transformers.

checker, it is disabled by default and can be enabled by the switch `-era`.

2.2 VC Generation from DSA

As we explained in the previous section, DSA is the input of the reachability analysis and thus it deserves special attention. Hence, in this section we formally define the DSA language and how a VC is obtained from a DSA program.

Before we proceed, we make several assumptions. In the rest of the paper we assume a first-order logic language for formulas and a theory T for the context of validity. We write $T \models f$ to denote that f is valid in the context of the theory T. The theory T expresses the *background predicate*, a (partial) axiomatization of the Java semantics.

We use f to denote a predicate represented as a logic formula where the free variables correspond to the predicate's arguments. In the following, by DSA we understand the language defined by the following grammar:

$$cmd := \textbf{skip} \mid \textbf{assume}\, f \mid \textbf{assert}\, f \mid cmd \,[\!]\, cmd \mid cmd; cmd$$

Additionally, we will use the following shorthands:

$$\textbf{if}\, C\, \textbf{then}\, B_1\, \textbf{else}\, B_2 \equiv (\textbf{assume}\, C; B_1) \,[\!]\, (\textbf{assume}\, \neg C; B_2)$$
$$\textbf{if}\, C\, \textbf{then}\, B \equiv \textbf{if}\, C\, \textbf{then}\, B\, \textbf{else skip}$$

Informally, the purpose of the **assume** f command is that, once the execution reaches this command, f can be assumed; if an execution trace reaches this command and f does not hold, that execution trace blocks. The purpose of the **assert** f command is that, once the execution reaches this command, f is checked and if it is invalid, an error occurs. The command $C_1 \,[\!]\, C_2$ represents a nondeterministic *choice* between the two commands and the command $C_1; C_2$ represents a *sequence*.

To formally define the semantics of DSA, we introduce two strongest postcondition predicate transformers — N and W. The predicate N propagates the *normal behavior* and the predicate W propagates the *wrong behavior*. Their semantics are captured by the following definition.

Definition 1. For the predicate transformers N and W defined as in Figure 3, we define the following:

1. For a precondition P and a command C, we say that C *goes wrong* if and only if $\mathrm{W}(P, C)$ is satisfiable, i.e.,
$$T \not\models \neg \mathrm{W}(P, C)$$
2. The *verification condition* for a program C is the following formula:
$$\neg \mathrm{W}(\mathit{true}, C)$$
3. The program C *conforms to its specification* if and only if its verification condition is valid:
$$T \models \neg \mathrm{W}(\mathit{true}, C)$$

Intuitively, the verification conditions expresses that no possible execution breaks any of the assertions.

An important property of this semantics is that a command with an unsatisfiable precondition does not go wrong.

Observation 1. *If $T \models \neg P$, then $T \models \neg W(P, C)$, for all predicates P and all commands C.*

This observation is not surprising since an unsatisfiable precondition states that the command in question should never be run according to its specification. What is less obvious is that this fact also comes into effect for a subcommand in a sequence of commands. For example, consider the sequence $C_1; (C_2; C_3)$. We can say that the postcondition of C_1 is a precondition of $C_2; C_3$. In particular, if $\mathrm{N}(\mathit{true}, C_1)$ is unsatisfiable, the whole sequence cannot go wrong because of C_2 or C_3. In other words, C_2 and C_3 are not checked. In such situations, an analysis relying on a strongest postcondition calculus does not provide any useful information about these subcommands. Moreover, such a scenario is most likely unintentional.

3. DEFINITION OF UNREACHABILITY

Informally, a command is unreachable if all the execution traces leading to it have an unsatisfiable normal behavior. To express this idea formally, this section defines the notion of unreachability in the context of the normal behavior (the predicate transformer N) and an acyclic control flow graph. Let $\mathcal{C}$ denote the subset of the DSA language consisting of the commands **skip**, **assume** f, and **assert** f.

Definition 2. A *control flow graph* is a tuple $\langle V, E, I, O, \mathcal{L} \rangle$, where V is the set of nodes, $E \subseteq V \times V$ is the set of edges, $I \subseteq V$ is the set of *entry nodes* and $O \subseteq V$ is the set of *exit nodes*. Nodes are labeled with commands by the function $\mathcal{L} : V \to \mathcal{C}$. Additionally, we require that entry nodes do not have parents, exit nodes do not have children, the graph is acyclic, and the set of nodes is finite.

The DSA maps to a subclass of control flow graphs, called *series–parallel* graphs [27], constructed as follows.

1. If C is one of **skip**, **assume** f or **assert** f, then it maps to $\langle \{n\}, \{\}, \{n\}, \{n\}, [n \mapsto C] \rangle$, where n is a fresh node
2. If C_1 maps to $\langle V_1, E_1, I_1, O_1, \mathcal{L}_1 \rangle$ and C_2 maps to $\langle V_2, E_2, I_2, O_2, \mathcal{L}_2 \rangle$ then,
   (a) $C_1; C_2$ maps to
   $$\langle V_1 \cup V_2, E_1 \cup E_2 \cup (O_1 \times I_2), I_1, O_2, \mathcal{L}_1 \cup \mathcal{L}_2 \rangle$$
   (b) and $C_1 \,[\!]\, C_2$ maps to
   $$\langle V_1 \cup V_2, E_1 \cup E_2, I_1 \cup I_2, O_1 \cup O_2, \mathcal{L}_1 \cup \mathcal{L}_2 \rangle$$

Once we have the control flow graph, the definition of unreachability is straight forward.

Definition 3. We define the *parents* and the *precondition* of a node in a control flow graph $G \equiv \langle V, E, I, O, \mathcal{L}\rangle$ as follows:

$$\text{parents}_G(n) \equiv \{p \in V \mid \langle p, n\rangle \in E\}$$

$$\text{pre}_G(n) \equiv \begin{cases} \mathit{true}, & \text{if } n \in I \\ \bigvee_{p\in\text{parents}_G(n)} \text{N}(\text{pre}_G(p), \mathcal{L}(p)), & \text{otherwise} \end{cases}$$

Definition 4. Node n is *semantically unreachable* in a control flow graph G if and only if $T \models \neg\, \text{pre}_G(n)$.

Whenever we use the term 'unreachable' (and 'reachable') we refer to semantic unreachability as defined above, not to the graph-theoretic notion.

3.1 How Unreachability Corresponds to DSA

To better understand the definition of semantic unreachability (Definition 4), we explore how the DSA command corresponds to its control flow graph. First observe that the DSA semantics (see Figure 3) has the following properties, where $\mathcal{B}$ is either N or W.

$$\mathcal{B}(P, (C_1 \,[\!]\, C_2); D) = \mathcal{B}(P, (C_1; D) \,[\!]\, (C_2; D)) \quad (1)$$

$$\mathcal{B}(P, D; (C_1 \,[\!]\, C_2)) = \mathcal{B}(P, (D; C_1) \,[\!]\, (D; C_2)) \quad (2)$$

$$\mathcal{B}(P, C_1 \,[\!]\, (C_2 \,[\!]\, C_3)) = \mathcal{B}(P, (C_1 \,[\!]\, C_2) \,[\!]\, C_3) \quad (3)$$

$$\mathcal{B}(P, C_1; (C_2; C_3)) = \mathcal{B}(P, (C_1; C_2); C_3) \quad (4)$$

By applying these equalities, any command can be rewritten into the choice between all its possible *execution traces*:

$$(C_1^1; C_2^1; \ldots; C_{l_1}^1) \,[\!]\, \ldots \,[\!]\, (C_1^m; C_2^m; \ldots; C_{l_m}^m)$$

where each C_i^j is neither the choice nor the sequence, while preserving the behaviors' semantics. The paths in the graph obtained from the process described above correspond to these execution traces. For both the normal and the wrong behavior, each behavior is a disjunct of the behaviors of these traces (see Figure 3); in particular the whole command goes wrong if and only if at least one of its traces goes wrong.

Let us consider a node n labeled with the command C. Then each of the paths going through n correspond to an execution trace of the form $C_{\text{pre}}; C; C_{\text{post}}$, where C_{pre} is a *prefix* of the pertaining execution trace. Let Pre be the set of all these prefixes, then the function $\text{pre}_G(n)$ (see Definition 3) returns the disjunct of normal behaviors of the prefixes, i.e., $\bigvee_{C_\text{p}\in\text{Pre}} \text{N}(\mathit{true}, C_\text{p})$.

Hence, Definition 4 captures our intuition that a command is unreachable if all the paths leading to it have an unsatisfiable normal behavior. In particular, if C is an assertion, the whole program cannot go wrong because of that assertion if it is unreachable as none of the traces leading to C can go wrong because of C (see Observation 1).

4. SCENARIOS OF UNREACHABLE CODE

In this section we discuss typical scenarios that result in unreachable code. In Section 6 we present how often these scenarios appear in practice. We start by showing several typical cases of discrepancies in the code or specifications. The last two subsections discuss unreachable code in the presence of loops.

4.1 Incoherence of Specification and Code

We present three kinds of unreachable code in Figure 4. The unreachableCode method contains unreachable code in the classic sense: the division by zero is not checked. It is most likely a bug in the user code. More subtle problems arise when we take into account annotations as well, as in the withPre method from Figure 1. An extreme example of an inconsistency in specifications is the method badSpec which has an unsatisfiable precondition. Such methods always pass extended static checking without reachability analysis.

A common case of unreachable code is related to the use of JML's `modifies` clause. Consider the methods modA, which promises to modify only a, and modAB, which can also modify b. Therefore, modAB should not be called from modA. ESC/Java2 models this by inserting an **assert** *false* before the call to modAB. This causes the rest of the assertions to be unreachable. This scenario is a specific instance of a general issue where an unsatisfiable asserted expression generates one warning and hides other warnings. (Assertions that are merely invalid but satisfiable do not hide other checks.)

In the introduction we have already mentioned that specifications of methods for which an implementation is not available are a common source of inconsistencies. Consider again the methods libraryFunc and useLibraryFunc in Figure 1. The body of the method useLibraryFunc is translated to DSA as follows:

C_1: **assert** $11 \geq 10$;
C_2: **assume** $r_1 = 11 \wedge r_1 < 10$;
C_3: **assert** $0 \neq 0$;
C_4: **assume** $RES = 1/0$

Here, the command C_1 represents the check for the precondition of libraryFunc and C_2 represents its postcondition. This is a general approach of translating method calls, preconditions are translated as asserts and postconditions as assumes. If the called method can modify the program state, the variables whose values may change need to be reset. The technique for reseting values of variables is called havocking and we will briefly describe it in Section 4.2. Nevertheless, in this particular case the modifies \nothing; clause in the specification of libraryFunc guarantees that it does not modify anything. The command C_3 checks that the division by 0 will not occur and the command C_4 stores the result of the division in a special variable RES modeling the method's return value. Apparently, the normal behavior $(11 \geq 10) \wedge (r_1 = 11 \wedge r_1 < 10)$ of $C_1; C_2$ is unsatisfiable. Recalling Definition 4, the commands C_3 and C_4 are unreachable. Hence, the method uselibraryFunc cannot go wrong because of the assertion C_3, i.e., it is not checked.

Because ESC/Java2 is a modular checker, method calls are always checked with respect to the specification of the called method and its implementation is ignored. This means that the situation described above would occur even if we did have an implementation for the method libraryFunc. If we had the implementation, however, we would uncover that the postcondition is not satisfiable during the check of that implementation (as an unsatisfiable postcondition cannot be established). See [6] for a detailed discussion about the pitfalls of specifications without implementation.

4.2 Safe Loop Desugaring

As we have described in Section 2, ESC/Java2 supports two modes of loop desugaring. In this section we discuss

```
//@ requires x > 0;
//@ requires x < 0;
int badSpec(int x, int y) {
   return 1/0;
}

int unreachableCode(int x) {
  if (x > 10)
    if (x < 5)
      return 1/0;
  return 0;
}
```

```
//@ modifies a, b;
void modAB() { ... }

//@ modifies a;
int modA() {
  modAB();
  return 1/0;
}
```

Figure 4: Examples of different types of unreachable code.

what information the reachability analysis provides in the safe desugaring mode. Please recall that the loop safe mechanism relies on loop invariants.

Since the reachability analysis does not depend on the way loops are desugared, we do not provide full account of this process. We illustrate the behavior on examples instead. Consider the following method where the user provided the invariant $i \geq 5$:

```
//@ requires i >= 5;
void infiniteLoop(int i) {
  //@ loop_invariant i >= 5;
  while (i >= 0) i = 5;
  //@ assert false;
}
```

In the loop safe mode this method is desugared as follows:

C_1: **assume** $i_0 \geq 5$;
C_2: **assert** $i_0 \geq 5$;
C_3: **assume** $i_1 = i'$;
C_4: **assume** $i_1 \geq 5$;
C_5: ((**assume** $i_1 \geq 0$; **assume** $i_2 = 5$;
assert $i_2 \geq 5$; **assume** *false*)
$[\!]$(**assume** $\neg(i_1 \geq 0)$);
C_6: **assert** *false*

The command C_1 represents the precondition and C_2 checks for the validity of the loop invariant before the loop. The command C_3 resets the value of i to a fresh value. This is called *havocking* and it models the fact that for an arbitrary iteration we do not know anything about the variables modified in the loop body except for what is in the loop invariant. In other words, havocking discards the information about these variables that was available before the loop. The command C_4 corresponds to the fact that the loop invariant holds before any iteration (knowing that it was established before the loop). The command C_5 represents a choice between termination of the loop and the loop body. More precisely, the left branch of the choice command models an arbitrary iteration of the loop, checks the loop invariant after the iteration, and blocks. The right branch of the choice command conditions further execution by the negation of the loop's guard.

Now we observe that the conjunct of the invariant and the negation of the guard $(i \geq 5) \wedge \neg(i \geq 0)$ is unsatisfiable. Therefore, the reachability analysis detects that the assertion at the end is unreachable.

The example above illustrates that the reachability analysis discovers that a loop does not terminate but only if the loop invariant is strong enough. Thus, it would be beneficial to combine the reachability analysis with techniques for loop invariant generation [10, 17, 23]. For example, consider the following excerpt of code:

```
int sum = 0;
for (int i = 0; i < 10; j++) sum += i;
//@ assert false;
```

The loop above does not terminate. If a technique for loop invariant inference is used, the user is likely to expect that the invariant $0 \leq i \wedge i \leq 10$ will be automatically inferred. Instead, however, the invariant $i = 0$ is inferred and the rest of the method is unchecked. Hence, the reachability analysis provides a warning about this bug.

4.3 Loop Unrolling

Apart from the safe desugaring discussed in the previous section, ESC/Java2 supports an unsound handling of loops called loop unrolling. This technique is parameterized by a constant L and reasons only about the scenarios when a given loop terminates in $0, 1, \ldots, L$ iterations. By following this approach, ESC/Java does not detect errors that may only happen when a loop is executed more than L times. The following schematically describes the result of an unrolling for $L = 2$:

```
while (C) {
   B
}
```
$\Rightarrow$
if C **then** B;
if C **then** B;
if C **then assume** *false*;

Execution traces that do not terminate in L iterations are modeled as blocking in the loop by the command **assume** *false*.

Loop unrolling contains a significant pitfall. If for all possible inputs the analyzed loop does not terminate within L iterations, the checker does not reason about the code following the loop.

Consider the following translation of a Java code to its DSA representation (for $L = 2$):

```
int i = 0;
while (i < 10)
   i++;
return 1/0;
```
$\Rightarrow$
C_1: **if** $0 < 10$ **then**
assume $i_1 = 0 + 1$;
C_2: **if** $i_1 < 10$ **then**
assume $i_2 = i_1 + 1$;
C_3: **if** $i_2 < 10$ **then**
assume *false*;
C_4: **assert** $0 \neq 0$;
C_5: **assume** $RES = 1/0$

We note that $T \models \neg\mathrm{N}(\mathit{true}, C_1; C_2; C_3)$. From Observation 1, it follows that $T \models \neg\mathrm{W}(\mathrm{N}(\mathit{true}, C_1; C_2; C_3), C_4; C_5)$. Therefore, the assertion C_4 cannot cause the program to go wrong since from the point of view of the checker that assertion is unreachable.

The analysis presented in this article detects that the code following the loop is not checked. Once the user is informed about it, he or she may either instruct ESC/Java2 to unroll the loop more times or may provide appropriate loop invariants and instruct ESC/Java2 to use safe desugaring.

5. THE ALGORITHM

We are given a directed acyclic flow graph in which we want to detect semantically unreachable nodes. An efficient algorithm is needed to make the analysis usable in practice. For that we need to (1) compute small prover queries, and (2) call the prover only a few times. Experimental data shows that the response time of the automated theorem prover used in these experiments (Simplify [12]) sharply increases when the size of the query exceeds a certain limit, which motivates (1). A prover call is on average hundreds times slower than any reasonable manipulation of the flow graph, which motivates (2).

The precondition of each node can be computed from Definition 3. If the implementation is memoized then the precondition will be represented as a directed acyclic graph (DAG) with $n-1$ nodes for $\vee$ and m nodes for $\wedge$, where n is the number of nodes and m is the number of edges in the flow graph. (Note that $\mathrm{N}(\mathrm{pre}_G(p), \mathcal{L}(p))$ may introduce at most one $\wedge$ operator, according to Figure 3.) Unfolding the DAG naively into a tree to send it to a prover often yields queries with exponential size. A simple way to obtain preconditions that produce queries with linear size is to introduce an auxiliary variable for each precondition, and then use it to express subsequent preconditions. But auxiliaries increase the query size. We can minimize the size of the formula by introducing auxiliaries only for subformulas of size S when they appear in P places and $PS - P - S \geq 2$. This transformation reduces the size of the queries dramatically: On our benchmarks it reduced by 90% the number of queries that are too big for the prover to process. This transformation exploits the series–parallel structure of the flow graph. Hence, the queries are roughly the same size as the normal behavior computed directly on the DSA as in [14].

The auxiliary variables can be defined using equivalence.

$$\big(a \Leftrightarrow f(b)\big) \wedge g(a, b) \tag{5}$$

Here b is a set of variables, a is the auxiliary variable, $f(b)$ is its definition, and $g(f(b), b)$ is the original formula. Now consider the alternative:

$$\big(a \Rightarrow f(b)\big) \wedge g(a, b) \tag{6}$$

It can be shown that (5) is satisfiable if and only if (6) is satisfiable, provided that g is monotonic in a, that is, $g(\mathit{false}, b) \Rightarrow g(\mathit{true}, b)$. We can make sure that that is the case by eliminating sharing only below the operators $\wedge$ and $\vee$. (Note that $\wedge$ and $\vee$ are the only operators introduced by the N predicate.) In practice, replacing (5) by (6) reduces the proving time to two thirds.

We say that the nodes of the flow graph that can be tracked back to Java code are *interesting*. The details of how to keep track from where in the Java code a DSA command comes from are outside the scope of this paper and can be found elsewhere [24]. For typical Java code there are less than 20 interesting nodes in most cases. Processing them takes negligible time, which is why later we shall concentrate on minimizing the number of prover queries. We contract the graph by keeping only the interesting nodes; we have an edge (u, v) in the contracted graph if in the original one there was a path from u to v with no other interesting node. This can be done in $O(mn)$ time with a slight modification of a DFS-based solution to the transitive closure problem. The contracted graph has a unique inital node denoted by i.

The key observation that allows us to have fewer prover calls than interesting nodes is that the information about node reachability can be propagated in the flow graph according to these rules: (1) we can infer that u is unreachable if all paths from i to u contain an unreachable node, and (2) we can infer that u is reachable if it dominates a reachable node v, that is, if all paths from i to v that do not contain unreachable nodes go through u. These rules are expressed in terms of paths, implying that we can use the propagation algorithm (Figure 5) on the original graph as well as on the contracted graph.

```
PROPAGATE-UNREACHABLE(u)
    label u as unreachable
    for each child v of u
      such that v has only unreachable parents
        do PROPAGATE-UNREACHABLE(v)


PROPAGATE-REACHABLE(u)
    label u as reachable
    if u has an immediate dominator d
       then PROPAGATE-REACHABLE(d)
```

Figure 5: Reachability propagation.

```
REACHABILITY-ANALYSIS()
    while there are unlabeled nodes
        do choose an unlabeled node u that has
              a maximal number of unlabeled dominators
           if the prover says that
              the precondition of u is satisfiable
            then PROPAGATE-REACHABLE(u)
            else use binary search with prover queries
                    to identify the farthest
                    unreachable dominator d of u
                 PROPAGATE-UNREACHABLE(d)
                 RECOMPUTE-DOMINATORS
                 if d has an immediate dominator d'
                    then PROPAGATE-REACHABLE(d')
```

Figure 6: The algorithm implementing the analysis.

We compute dominators ignoring nodes already marked as unreachable using the simple algorithm of Cooper [8], which works in $O(mn)$ time for DAGs. The critical part that makes

our implementation fast in practice is the heuristic used to decide for which node we query the prover.

In the case that all nodes are reachable and interesting the greedy algorithm (Figure 6) is optimal, because the prover must be called for all the leafs of the (immediate) dominator tree. In practice the performance is good. We have run ESC/Java2 on its front-end (`javafe`) which contains 1890 methods and is one of the largest coherent set of JML-annotated code available. The total running time is 31589 seconds (almost 9 hours), out of which 34.8% is spent in the reachability analysis, out of which 99.8% is spent in the prover. The total number of leafs in the dominator trees is 3256 and the number of prover calls is 3351. The average number of nodes in the flow graph is a few hundred and in the contracted flow graph it is 10. For this benchmark we used the default loop desugaring in ESC/Java2, which is unrolling once.

6. CASE STUDY

As described in the previous section, we have tested the analysis on the ESC/Java2 front-end, the `javafe` package. The package contains 217 classes.

We have found 5 inconsistencies in the specifications of the JDK that are not reported without the reachability analysis. More details can be found in the ESC/Java2 bug-tracker[2] under the bugs #595, #550, #568, #549, #545. We found one more inconsistency in the JDK specification which was due to the incorrect handling of a JML feature *informal comment* by ESC/Java2. ESC/Java2 treats an informal comment as *true*, this is harmless in most cases (such as `requires (* is upper-case *)`) but for example, `ensures \result <=> (* is upper-case *)` likely results in an unintended specification (see bug #547).

ESC/Java2's repository contains handcrafted tests to detect inconsistencies in the JDK specifications. These tests did not detect the problems uncovered by the reachability analysis because they are not exhaustive. We should note that fixing these problems involved a tedious process of narrowing down the set of inconsistent annotations. This effort, however, was justified by the wide usage of these specifications.

In 1 case a **catch**-block was unreachable because it was catching an exception that was not declared in any of the specifications of the methods called in the **try**-block (see documentation for the `signals_only` pragma).

An incorrect use of the `modifies` clause (as in Figure 4) hiding the rest of the potential warnings appeared 9 times. Warnings hiding subsequent code appeared 6 times. The case of unreachable code resulting from loop unrolling, as discussed in Section 4.3, appeared 4 times. In 9 cases the informal comments indicated that the author was aware that the code is unreachable. The user can mark such code with the `unreachable` pragma and then the analysis does not warn about it. We detected only one case of unreachable code in the classical sense.

In several cases the unreachability was due to the unsound modeling of the `modifies \everything;` pragma. This pragma is the default annotation if no `modifies` clause is provided. Whenever a method with the `modifies \everything;` annotation is called, ESC/Java2 does not consider the potential state change. Therefore, the code that we have found is actually executed. Nevertheless, ESC/Java2 does not check that code, thus the warnings provided by the analysis are not spurious.

In the remaining 12 cases we were not able to precisely identify the source of the problem. Nevertheless, we suspect that the source lies in inconsistent specifications of classes inside the `javafe` package. Such inconsistencies are very hard to pinpoint as they involve object invariants in a class hierarchy.

[2] `http://sort.ucd.ie/tracker/?group_id=97`

7. RELATED WORK

Traditionally, unreachable code is detected by techniques based on data flow analysis or abstract interpretation [9]. These techniques are generally known under the term *dead code elimination* [26] and are used for code optimization. To our knowledge, automated theorem proving is not used in mainstream compilers. Interactive theorem proving, however, is used to show properties of code optimizations. For example, Blech et al. [3] applied the higher-order theorem prover Isabelle/HOL to mechanically prove that a code optimization based on dead code elimination is semantics preserving.

Another stream of research related to our work is focused on reasoning about specifications for which there is no implementation available. Chalin [6] describes an enhancement of ESC/Java2 that checks 'definedness' of specifications. An example of a partial specification is `requires a.x == 0;` since it does enforce `a` to be non-**null**. As in the case of our work, this technique is fully automated.

Bouquet et al. [4] use a constraint solver to *animate* specifications. Basically, specification animation provides a way to debug specifications without implementation. The animating system maintains an abstract state and the user can ask the system what happens to that state if a certain method is called. Using this technique, it is possible to uncover that a sequence of method calls necessarily lead to an inconsistent state.

The term reachability analysis is used in related areas in a slightly different sense. In model checking it denotes the analysis that searches for reachable states of the given state space [1]. In heap analysis the reachability analysis is done on the reference graph [7].

8. SUMMARY AND FUTURE WORK

We devised the theoretical underpinnings of reachability analysis for annotated code, implemented it efficiently, and classified the bugs that it helps to find. We intend to adapt it for BoogiePL [11], whose flow graphs are not necessarily series–parallel.

We pose two open problems related to this analysis.

Provide better warnings. As the case study shows, although our analysis uncovers real bugs, they are often hard to track down. The warning message should also pinpoint the likely locations causing code to be unreachable, not only the location of the unreachable code. Even better, the warning should also classify the problem, for example by saying that it is a 'loop unrolling' problem if that is the case.

Optimize VCs and prover queries. The reachability analysis suggests that one VC per method might not be optimal, for example because it includes all the unreachable code. In general, what is an optimal strategy for querying the prover for the correctness of a method, given its flow graph?

9. ACKNOWLEDGMENTS

This work is funded by Science Foundation Ireland under grant number 03/CE2/I303-1, "LERO: the Irish Software Engineering Research Centre" and by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under the IST-2005-015905 MOBIUS project. The article contains only the authors' views and the Community is not liable for any use that may be made of the information therein.

10. REFERENCES

[1] P. Abdulla, P. Bjesse, and N. Een. Symbolic reachability analysis based on SAT-solvers. *Tools and Algorithms for the Analysis and Construction of Systems (TACAS)*, 2000.

[2] M. Barnett, K. Leino, and W. Schulte. The Spec# programming system: An overview. In *Proceeding of CASSIS 2004*, volume 3362 of *Lecture Notes in Computer Science*. Springer–Verlag, 2004.

[3] J. O. Blech, S. Glesner, and J. Leitner. Formal verification of dead code elimination in Isabelle/HOL. In *SEFM '05: Proceedings of the Third IEEE International Conference on Software Engineering and Formal Methods*, 2005.

[4] F. Bouquet, F. Dadeau, B. Legeard, and M. Utting. Symbolic animation of JML specifications. In *Proceedings of Formal Methods, International Symposium of Formal Methods (FM 2005)*, Lecture Notes in Computer Science. Springer–Verlag, 2005.

[5] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, Feb. 2005.

[6] P. Chalin. Early detection of JML specification errors using ESC/Java2. In *Proceedings of the Workshop on the Specification and Verification of Component-Based Systems (SAVCBS)*. ACM Press, Nov. 2006.

[7] S. Chatterjee, S. Lahiri, S. Qadeer, and Z. Rakamaric. A reachability predicate for analyzing low-level software. In *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2007.

[8] K. D. Cooper, T. J. Harvey, and K. Kennedy. A simple, fast dominance algorithm, 2001. Available online at `www.cs.rice.edu/~keith/EMBED/dom.pdf`.

[9] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press.

[10] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, 1978.

[11] R. DeLine and K. R. M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical report, Microsoft Research, 2005.

[12] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *Journal of the Association of Computing Machinery*, 52(3):365–473, 2005.

[13] C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Press, 2002.

[14] C. Flanagan and J. B. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *Conference Record of the 28th Annual ACM Symposium on Principles of Programming Languages*, pages 193–205. ACM, Jan. 2001.

[15] L. Friendly. The design of distributed hyperlinked programming documentation. *IWHD*, 95:151–173, 1995.

[16] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969.

[17] M. Janota. Assertion-based loop invariant generation. In *Proceedings of the 1st International Workshop on Invariant Generation (WING '07)*, June 2007.

[18] J. R. Kiniry and D. R. Cok. ESC/Java2: Uniting ESC/Java and JML: Progress and issues in building and using ESC/Java2 and a report on a case study involving the use of ESC/Java2 to verify portions of an Internet voting tally system. In *Construction and Analysis of Safe, Secure and Interoperable Smart Devices: International Workshop, CASSIS 2004*, volume 3362 of *Lecture Notes in Computer Science*. Springer–Verlag, Jan. 2005.

[19] D. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.

[20] C. Krueger. Software reuse. *ACM Computing Surveys*, 24(2), 1992.

[21] G. T. Leavens, A. L. Baker, and C. Ruby. *Behavioral Specifications of Business and Systems*, chapter JML: A Notation for Detailed Design, pages 175–188. Kluwer Academic Publishing, 1999.

[22] G. T. Leavens, K. R. M. Leino, and P. Müller. Specification and verification challenges for sequential object-oriented programs. *Formal Aspects of Computing*, 19, 2007.

[23] K. R. M. Leino and F. Logozzo. Loop invariants on demand. In *Proceedings of The Third Asian Symposium on Programming Languages and Systems (APLAS 2005)*, volume 3780 of *Lecture Notes in Computer Science*. Springer–Verlag, 2005.

[24] K. R. M. Leino, T. Millstein, and J. B. Saxe. Generating error traces from verification-condition counterexamples. *Science of Computer Programming*, 55(1-3):209–226, 2005.

[25] K. R. M. Leino, J. B. Saxe, and R. Stata. Checking Java programs via guarded commands. Technical Note 1999-002, Compaq SRC, May 1999.

[26] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer–Verlag, 1999.

[27] J. Valdes, R. E. Tarjan, and E. L. Lawler. The recognition of Series–Parallel digraphs. *Proceedings of the eleventh annual ACM symposium on Theory of computing*, pages 1–12, 1979.

Faithful mapping of model classes to mathematical structures

Ádám Darvas
ETH Zurich
adam.darvas@inf.ethz.ch

Peter Müller
Microsoft Research
mueller@microsoft.com

Abstract

Abstraction techniques are indispensable for the specification and verification of functional behavior of programs. In object-oriented specification languages like JML, a powerful abstraction technique is the use of model classes, that is, classes that are only used for specification purposes and that provide object-oriented interfaces for essential mathematical concepts such as set or relation.

While the use of model classes in specifications is natural and powerful, they pose problems for verification. Program verifiers map model classes to their underlying logics. Flaws in a model class or the mapping can easily lead to unsoundness and incompleteness.

This paper proposes an approach for the faithful mapping of model classes to mathematical structures provided by the theorem prover of the program verifier at hand. Faithfulness means that a given model class semantically corresponds to the mathematical structure it is mapped to.

Our approach enables reasoning about programs specified in terms of model classes. It also helps in writing consistent and complete model-class specifications as well as in identifying and checking redundant specifications.

Categories and Subject Descriptors D.2.1 [*Software Engineering*]: Requirements/Specifications—Methodologies; D.2.4 [*Software Engineering*]: Software/Program Verification—Formal methods, Programming by contract

General Terms Verification

Keywords specification, verification, abstraction, model classes, isomorphism, Java Modeling Language

1. Introduction

Abstraction is indispensable for the functional specification and verification of object-oriented programs. Without abstraction, types with no implementation (i.e. interfaces or abstract classes) cannot be specified. Abstraction is also necessary to support subtyping and information hiding.

One way of expressing data abstraction in specification languages is by relating implementations to corresponding mathematical structures such as sets and tuples. This approach was pioneered by the Larch project [10], which advocated two-tiered specifications consisting of a contract and a theory providing the mathematical structures.

Sixth International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2007), September 3-4, 2007, Cavtat near Dubrovnik, Croatia.


```
package java.util;
//@ import org.jmlspecs.models.JMLObjectSet;

public interface Set extends Collection {
  //@ public model instance JMLObjectSet _set;

  /*@ also
    @   public normal_behavior
    @     ensures contains(o);
    @*/
  public boolean add(Object o);

  /*@ also
    @   public normal_behavior
    @     ensures \result == _set.has(o);
    @*/
  /*@ pure @*/ public boolean contains(Object o);

  // other constructors and methods omitted
}
```

Figure 1. Specification of type `Set` using model class `JMLObjectSet` defined in JML's model library. JML annotation comments start with an at-sign (@). Keyword `also` expresses that the given specification extends the specification given in supertype `Collection`. The import declaration allows one to refer to the model class. We omit `nullable` annotations for brevity.

The Java Modeling Language (JML) unifies these tiers to simplify the development of specifications [4]. Instead of using a separate language to describe mathematical structures, JML describes them in an object-oriented manner through *model classes*. These classes contain only pure (side-effect free) methods. Therefore, they can be used in specification expressions.

Figure 1 shows the use of model class `JMLObjectSet` for the specification of the interface `Set`. The model class, through its pure methods, provides access to a mathematical set that contains objects. In order to use the model class, a model field `_set` is declared. This field is used for specification purposes only and is supposed to represent the abstraction of an instance of type `Set`.

One can specify `Set`'s method `contains` in an abstract way using the model field and pure method `has` declared in model class `JMLObjectSet`. Given a concrete implementation of `Set` one would define the relation (using JML's `represents` clause) between the public model field `_set` and the private internal structure.

While model classes are useful for specification purposes, they pose problems for verification. Program verifiers have to encode model classes in the underlying theorem prover. This can be done by encoding pure methods and their contracts as uninterpreted function symbols and axioms, respectively [6, 5, 12]. However, this approach is not optimal for model classes because the tactics of theorem provers are optimized for the prover's theories rather than encodings of JML model classes. Moreover, it is difficult to

ensure soundness of such encodings, especially in the presence of recursive specifications [6].

To overcome these problems, previous work [3, 13, 14] proposes to map model classes and their pure methods directly to theories of the theorem prover at hand. For instance, a method `contains` of a model class could be mapped to the $\in$ operator of the theorem prover. However, the existing work only discusses the mapping of method signatures, but ignores their contracts. With this approach, the meaning of `contains` is given by the definition of $\in$, and not by its contract. This is problematic if there is a mismatch between the contract and the semantics of the operation given by the theorem prover: static program verifiers might produce results that come unexpected for programmers relying on the model class contract. The results may also vary between different theorem provers, which define certain operations slightly differently. Moreover, runtime assertion checking might diverge from static verification if the model class implementation used by the runtime assertion checker is based on the model class contract.

In this paper, we show how model classes can be mapped to theorem provers without semantic mismatches. The main contribution of our work is a technique for proving that the mapping of a model class to a mathematical structure defined by the theorem prover is faithful. *Faithfulness* means that the model class and the structure indeed correspond to each other in their properties. To show faithfulness, we prove formally that the mapping is consistent and complete. *Consistency* means that everything that can be proved using the contracts of the model class can also be proved using the corresponding structure of the theorem prover. *Completeness* means that everything that can be proved using the structure defined by the theorem prover can also be proved using the contracts.

Our approach leads to important results beyond semantical correspondence. Model class contracts are complex and can easily get inconsistent, which can lead to unsound reasoning. Showing that a model class can be mapped consistently to a mathematical structure proves that the model class contract itself is consistent (provided that the structure is well-defined). In fact, our case study discovered an inconsistent specification in one of the most basic model classes of JML.

This shows that proving faithfulness of mappings helps in writing better specifications for model classes by making them consistent and complete. Our approach can also be used to identify redundant parts of specifications as well as to check whether specifications marked as redundant are indeed derivable from non-redundant specifications. These capabilities further improve the quality of model-class specifications.

Throughout the paper we will use JML [15] as specification language and Isabelle [18] as target theorem prover. This choice was made due to the characteristics of the static program verification tool JIVE [16] that we are working on. However, the presented approach is applicable to any combination of specification languages and theorem provers, for instance, Eiffel [21] and Coq [1].

Our approach does not yet have tool support. All steps in the case study were performed manually. In Section 6, as future work we briefly mention areas where tool support could greatly help.

The rest of the paper is structured as follows. Section 2 introduces model class `JMLObjectSet`, the class we use throughout this paper to illustrate our approach. Section 3 presents our solution for the faithful mapping of model classes to mathematical structures defined by a theorem prover. Section 4 presents a case study we performed on model class `JMLObjectSet`. Section 5 gives an overview of related work and in Section 6 we conclude.

2. Running example

As running example, we take model class `JMLObjectSet`, which is part of the model library of the JML distribution. It models a set of objects. That is, it provides the usual operations of mathematical sets, and equality of elements is based on Java's reference equality ("`==`"). Figure 2 presents the class with the constructors and methods that we discuss in this paper.

The class and, thus, all its methods are pure. Methods that return `JMLObjectSet`s (for instance, `union`) do not mutate their receiver objects but return newly created `JMLObjectSet`s. In accordance with the JML semantics, all reference type arguments and return values are considered to be non-null.

The class is specified by an *equational theory* and by *method specifications*:

1. The equational theory is an object invariant expressed in terms of the static pure model method `equational_theory` which has to return true for every non-null `JMLObjectSet` instance `s2`, and objects `e1` and `e2`. The method has a large normal behavior specification case containing equations written in the style of algebraic laws. Figure 2 shows a sample equation defining method `union`.
2. Method specifications consist of pre- and postconditions attached to constructors and methods of the model class. Modifies clauses are not needed since all methods are pure. As an example, the specification of method `union` is given on Figure 2.

We follow the proposal of Leavens et al. [13] and Charles [3], and consider model classes to be final and unrelated to Java's type hierarchy rooted in type `Object`. This prevents problems related to inheritance, method overriding, and dynamic dispatch. In the realm of model classes, these restrictions seem acceptable since model classes are supposed to describe elementary mathematical concepts and to be used only for specification purposes.

3. Faithful mappings

In this section we present our solution for proving that the mapping of a model class M to a structure S (defined by some datatype or theory) is faithful. That is, there is a semantic correspondence between M and S, namely, they are *isomorphic*.

The process consists of three stages. In the first stage, we specify the mapping of M to S by a new JML clause, `mapped_to`. Then we prove consistency and completeness of the specified mapping in the second and third stages, respectively. In this section we present the details of these stages.

3.1 Specifying the mapping

In the first stage, one has to decide how to map model class M. That is, one has to specify (1) to what structure S is the model class mapped; and (2) to which function symbols of S are the methods of the model class mapped.

Figure 2 demonstrates a possible mapping of model class `JMLObjectSet`. The model class is mapped to Isabelle's `HOL/Set` theory [19], specifically to type "*'a set*". In Isabelle, *'a* is a type variable which gives rise to polymorphic types [18]. This mapping is specified by the new specification construct `mapped_to`. The first parameter specifies the target environment, the second the target context, and the third the specific type in the context to which the model class is mapped.

The `mapped_to` clause attached to the class determines the context and type to which the methods of the model class will get mapped. The mapping of the methods is specified by `mapped_to` clauses attached to the methods. For instance, method `has` is mapped to Isabelle's set membership ":". The first parameter is again the target environment, the second specifies the way the model class method is mapped to some term in the target context. The second parameter typically mentions function symbols of the target context as well as parameters (including the receiver) of

```
package org.jmlspecs.org;

//@ mapped_to("Isabelle","HOL/Set", "'a set");
public final /*@ pure @*/ class JMLObjectSet {

   /*@ public invariant
     @   (\forall JMLObjectSet s2; s2 != null;
     @     (\forall Object e1, e2; ;
     @       equational_theory(this, s2, e1, e2) ));
     @*/

   /*@ public normal_behavior
     @     ensures \result <==>
     @       (s.union(s2)).has(e1) ==
     @         (s.has(e1) || s2.has(e1));
     @   also
     @     ...
     @ static public pure model boolean
     @   equational_theory(JMLObjectSet s,
     @     JMLObjectSet s2, Object e1, Object e2);
     @*/

   //@ mapped_to("Isabelle","{}");
   public JMLObjectSet();

   //@ mapped_to("Isabelle","{e}");
   public JMLObjectSet (Object e);

   //@ mapped_to("Isabelle","elem : this");
   public boolean has(Object elem);

   //@ mapped_to("Isabelle","this = s2");
   public boolean equals(Object s2);

   //@ mapped_to("Isabelle","this = {}");
   public boolean isEmpty();

   public int int_size();

   //@ mapped_to("Isabelle","this <= s2");
   public boolean isSubset(JMLObjectSet s2);

   //@ mapped_to("Isabelle","this < s2");
   public boolean isProperSubset(JMLObjectSet s2);

   //@ mapped_to("Isabelle","SOME x. x : this");
   public Object choose();

   /*@ public normal_behavior
     @   ensures
     @     (\forall Object e; ;
     @       \result.has(e) <==>
     @         this.has(e) || (e == elem));
     @*/
   //@ mapped_to("Isabelle","insert elem this");
   public JMLObjectSet insert(Object elem);

   //@ mapped_to("Isabelle","this - {elem}");
   public JMLObjectSet remove(Object elem);

   /*@ public normal_behavior
     @   ensures
     @     (\forall Object e; ;
     @       \result.has(e) <==>
     @         this.has(e) || s2.has(e));
     @*/
   //@ mapped_to("Isabelle","this Un s2");
   public JMLObjectSet union(JMLObjectSet s2);
}
```

Figure 2. Model class `JMLObjectSet` containing the signatures of members we consider in this paper. The proposed mapping of the class and its members to Isabelle is given by the `mapped_to` clause. The object invariant, a fragment of the equational theory and two sample method specifications are given too. Other specification elements are omitted.

the method being specified. Note, however, that we permit arbitrary terms of the target context; this flexibility allows us to specify mappings even if the target theorem prover does not provide a structure that directly corresponds to the model class being mapped.

To support multiple theorem provers, multiple `mapped_to` clauses may be attached to the model class and its methods. This is needed since different theorem provers provide different theories with different function symbols and syntax for the same functionality. Thus, the isomorphism proof has to be carried out in every target theorem prover specified in `mapped_to` clauses.

Important to note is that the mappings need not be specified by programmers who are typically not familiar with theorem provers and their theories and syntax. The mappings can be specified by the author of a model class or by the team which performs the verification.

3.2 Consistency

Once the mappings are specified, their faithfulness has to be proven. For each theorem prover, this proof needs to be carried out only once. Afterwards, any verification system can make use of the specified mappings to handle model classes in specifications. In this section, we describe how to prove consistency of the mapping, that is, we prove that the properties of model class M (as specified by its contracts) can be derived from the properties of structure S (as defined by axioms, definitions, theorems etc.).

In order to prove consistency, one has to encode the method specifications and invariants of M in the language of S based on the `mapped_to` clauses and prove the resulting formulas using the properties of S. In fact, not all method specifications have to be translated and proved but only the ones that specify the *normal behavior* of a given method [15]. Other method specifications describe situations when the method might throw exceptions which is not of interest for the isomorphism proof.

In the sequel, we use the term *relevant specification element* to refer either to an invariant or to a normal-behavior method specification of a model class. Every relevant specification element s_M in M needs to be translated and proved as follows:[1]

1. (a) If s_M is a method specification of some method m with precondition pre and postcondition $post$ then it is treated as a formula of the form "$pre \Rightarrow post$" which is universally quantified over all parameters (including the implicit receiver) of m.
   (b) Occurrences of every method call to some method m have to be replaced by the term prescribed in the `mapped_to` clause of method m. For simplicity, we assume that JML's logical operators are also method calls with implicit mappings to the underlying theorem prover (e.g., JML's `==>` operator is mapped to logical implication).[2]
   (c) If s_M is a method specification of some method m, then in the postcondition all occurrences of `\result` (and `this` if m is a constructor) have to be replaced by the term prescribed in the `mapped_to` clause of method m.

2. The formula has to be turned into a lemma and proved in the theorem prover specified by the `mapped_to` clause using the axioms, definitions, theorems, etc. of S.

We demonstrate this process on `JMLObjectSet`'s `insert` method. Its method specification is presented on Figure 2.

[1] We ignore ghost fields in this paper. They can be handled by mapping a model class M with n ghost fields to a $n+1$-tuple, where the first component represents the structure for M and the other components represent the state of the ghost fields [17].

[2] Proving correspondence of logical operators is out of the scope of this paper.

In step 1(a) the postcondition gets universally quantified over the parameters of `insert`: $this$ and $elem$. In step 1(b) the two method calls on `has` get replaced by Isabelle's set membership operator ":" as prescribed by the `mapped_to` clause of `has` in Figure 2. This yields terms "$e : \texttt{\textbackslash result}$" and "$e : this$". Additionally, the logical operators `\forall`, `<==>`, `||`, and `==` get replaced by the corresponding Isabelle operators $\forall$, $=$, $\vee$, and $=$, respectively. Step 1(c) replaces `\result` by "$insert\ elem\ this$" as prescribed by the `mapped_to` clause of method `insert`. This yields the following formula:

$$\forall\, this, elem.\ \forall e.\\ (e : (insert\ elem\ this)) = ((e : this)\ \vee\ (e = elem))$$

In step 2 the formula is turned into a lemma in Isabelle. Its proof can be completed automatically by the **auto** tactic. This is not surprising since theorem provers like Isabelle are typically well-equipped with theorems over elementary structures.

Completing this stage successfully for every relevant specification element in model class M gives us the guarantee that whatever can be proven using the properties of M can be proven using S, too.

An important consequence of this result is that the specification of M is consistent (i.e., free of contradictions) provided S is consistent. Since structures like Isabelle's `Set` are defined using conservative extensions and have been reviewed by many people, it is rather unlikely that they contain inconsistencies. In other words, in this stage we prove that Isabelle's `Set` theory is a *model* for model class `JMLObjectSet`. By exhibiting this model we prove that using `JMLObjectSet`'s specification does not lead to unsoundness.

This is also an interesting result concerning the use of pure methods in specification expressions. As we have shown earlier [6], the use of pure methods in specifications can easily lead to unsoundness. The solution we proposed in our earlier work [6] to prevent unsoundness is to exhibit a witness for showing that the specification of the method is satisfiable. However, the solution is not applicable for recursive specifications. With the approach presented above, recursive specifications do not pose any problems.

As the example of method `insert` suggests, proving this stage may be fully automated: (1) the lemma was generated following three simple steps performing syntactic replacements based on `mapped_to` clauses, and (2) the lemma was proved without any user interaction using a powerful tactic of Isabelle.

3.3 Completeness

In the third stage, we complete the isomorphism proof by showing completeness of the mapping, that is, that the properties of structure S can be derived from the properties of model class M. The proof procedure is as follows:

1. Each member m of M is turned into a function symbol $\hat{m}$ and its signature is declared based on m's signature.
2. Each relevant specification element s_M in M is turned into an axiom as follows:
   (a) If s_M is a method specification with precondition pre and postcondition $post$ attached to method m then it is treated as formula "$pre \Rightarrow post$" universally quantified over all parameters of m.
   (b) Occurrences of method calls on some method m have to be replaced by function applications of the corresponding function symbol $\hat{m}$. Additionally, JML's logical connectives have to be replaced by the corresponding connectives of the theorem prover.
   (c) If s_M is a method specification of some method m, then in the postcondition all occurrences of `\result` (and `this` if m is a constructor) have to be replaced by function applications of function symbol $\hat{m}$.
3. A lemma is generated from every axiom and definition s_S of S by replacing all occurrences of function symbols in s_S by the corresponding function symbols declared in step 1. Correspondence is based on the `mapped_to` clauses.
4. The lemma has to be proven using the axioms generated in step 2.

As an example, we show this procedure for Isabelle's definition of proper subsets. In the first step, the signature of $\hat{isProperSubset}$ is declared based on the signature of method `isProperSubset`:

$$\hat{isProperSubset} : {'a\ set}\ \times\ {'a\ set} \Rightarrow bool$$

The second step is based on the specification of the method. For demonstration purposes, this time we take the specification prescribed by the invariant, i.e. the equation given in the specification of method `equational_theory`:

```
s.isProperSubset(s2) == (s.isSubset(s2) && !s.equals(s2))
```

Since the equation is part of the method specification of method `equational_theory`, first it gets quantified over its parameters: s, $s2$, $e1$, and $e2$. Then method calls on `isProperSubset`, `isSubset` and `equals` are turned into the function applications $\hat{isProperSubset}(s, s2)$ $\hat{isSubset}(s, s2)$ and $\hat{equals}(s, s2)$, respectively. Additionally, the logical operators are mapped. The resulting formula is turned into the following axiom:

$$\forall\, s, s2, e1, e2.\ \hat{isProperSubset}(s, s2) = \\ (\hat{isSubset}(s, s2) \wedge \neg \hat{equals}(s, s2)) \qquad (1)$$

In step 3, we take the definition of proper subsets from Isabelle's theory [19]:

```
psubset_def: "A < B == (A::'a set) <= B & ¬ A=B"
```

and translate it to the following lemma:[3]

$$\forall\, A, B.\ \hat{isProperSubset}(A, B) = \\ (\hat{isSubset}(A, B) \wedge \neg \hat{equals}(A, B))$$

In step 4, the lemma needs to be proven using only the axioms defined in step 2. The proof is trivial since axiom (1) (derived from the equational theory) is equivalent to the lemma.

Note that theorems of S need not be turned into lemmas since theorems are properties that are derived from definitions and axioms of S. However, it is important that all axioms and definitions are turned into lemmas, including the ones that do not appear in the textual representation of S. For instance, Isabelle supports inductively defined sets for which the tool generates fixed point definitions and proves several lemmas about them [18]. In such cases the artefacts introduced "under the hood" need to be turned into lemmas too. Theorem provers typically make these artefacts available for users, for instance, Isabelle can be queried to show them and PVS [22] generates separate files for them.

Proving this stage guarantees that whatever can be proved using the axioms, definitions and theorems of Isabelle's `Set`, can also be proved using JML's `JMLObjectSet`. An interesting consequence is that we have proved that the axiom system extracted from the specifications of `JMLObjectSet` is complete relative to Isabelle's theory of `Set`. Since Isabelle structures like `Set` are heavily used

[3] Isabelle definitions are implicitly universally quantified over variables that are not bound by quantifiers that appear explicitly.

in formalizations and proofs, one can be sure that they contain the most important properties of the structure.

The generation of lemmas (step 3) for this stage is not as trivial as for the consistency proof. The `mapped_to` clauses specify the mappings from M to S, which is the opposite direction of this stage. The mapping from S to M is not necessarily unique. For instance, the `=` operator of S is typically used in the `mapped_to` clauses of several methods of M, which makes it difficult to choose automatically the appropriate mapping.

Furthermore, proving the lemmas (step 4) is also less trivial than in the consistency proof. First, even the application of automated tactics typically requires one to manually select the set of axioms to be used for proving a given lemma because selecting all axioms might cause the tactic to loop. Second, the specifications of the model class may be too weak to verify some axioms or definitions of the structure. In this case, the missing specifications need to be identified and added to the model class. Thus, it seems that the automation of this stage can, in general, only be partial and manual intervention is needed for its completion. However, the effort is justified by the increased quality of the model class specification.

3.4 Summary

Successful completion of the three stages described above guarantees that model class M and structure S are isomorphic. This property confirms that the mappings prescribed by the `mapped_to` clauses were semantically correct.

The most important property from the consistency proof is that the axiom system extracted from the model class is consistent, thus its usage cannot lead to unsoundness. This is obviously a crucial property for every verification system. For this stage, the generation and proving of lemmas seem to be automatable. Failing to prove a lemma most probably indicates an error in the model class contract.

The most important result of the completeness proof is that the model class expresses the properties of the mathematical structure. This is important in order to prevent mismatches between the property one wants to express in a specification and the property one actually proves during the verification process. As noted above, the generation and proving of lemmas is not as trivial as for consistency.

Once both directions are successfully proved, method calls can be directly translated to the corresponding function applications without being worried about soundness issues or differences in the semantics of related methods and function symbols.

An interesting side-effect of the proposed proof technique is that redundant specifications can be discovered in the model class. If an axiom is never used in the completeness proof then the specification element from which the axiom was derived is redundant in the model class.

4. Case study

In this section, we demonstrate our approach for the model class `JMLObjectSet` by describing in detail the process of proving faithfulness with Isabelle's `HOL/Set` theory. We highlight the interesting observations and results of the case study.

We considered 17 members of the model class: 2 constructors, 9 query methods, and 6 methods that create new `JMLObjectSet` instances. These were all the members that remained after the simplifications described in the next section. All proofs were carried out in Isabelle. The proof scripts contained a total of ca. 380 LOC without comments and empty lines. Consistency of the mapping was proven in ca. 100, completeness in ca. 110 LOC. Equivalence of the equational theory and the method specifications (see Section 4.5) was proven in ca. 170 LOC. All proof scripts were written manually.

4.1 Simplifications

Since we were interested in the mapping of `JMLObjectSet` and its methods to an Isabelle theory, we first removed all methods that provided object-oriented features irrelevant for the mapping of the model class. These methods included, for instance, `clone`, `singleton`, `hashCode`, and `toString`. In our opinion, such methods need not be part of model classes if one thinks of them as mathematical structures.

As a next step we removed all implementation details. This included all method bodies, and members and specifications not visible for clients. Additionally, we removed all public members that were only used in informal specifications or provided only syntactic sugar. As mentioned in Section 3.2, only method specification that describe normal behavior need to be treated by our approach. Thus, we removed all other method specification cases. In order to keep our case study comprehensible, we removed ghost fields from the model class together with all specification expressions that referred to them.

To focus on the main ideas of this paper, we decided not to handle members that referred to non-primitive types other than `JMLObjectSet`. For instance, constructors that take as argument a node of a singly-linked list from which a `JMLObjectSet` is created, or methods that convert `JMLObjectSet`s to other model or non-model types. The handling of these kinds of members is possible once one has provided a mapping for the types mentioned in their signatures.

4.2 Division of specifications

We analyzed the specification of `JMLObjectSet` and found that the equational theory and method specifications contained a lot of redundancy. Many properties of the model class were attempted to be expressed both by the equational theory and by method specifications. We illustrate this by method `union`. The equation defining the method in the equational theory and its method specification is given on Figure 2. It is easy to see that after proper substitutions the two specifications express the same property.

Thus we decided to split specifications into two parts: one containing only the equational theory and the other containing only the method specifications. This allowed us to analyze their relation, as discussed in Section 4.5.

We note that it is not always necessarily the case that the equational theory of a model class and its method specifications contain so much redundancy. There are, for instance, JML model classes that specify the behavior of the class in great majority by method specifications (e.g., `JMLObjectToObjectRelation` and `JMLValueValuePair`). Thus, in general, faithfulness of a model class to some structure should be proven using both the equational theory and the method specifications together.

4.3 Specifying the mapping

The next step was the specification of the mapping of the model class and its methods. The resulting mapping to Isabelle's higher-order set theory `HOL/Set` is shown in Figure 2.

The mapping of the different methods of the model class was mostly straightforward. Here we mention three interesting cases. Method `choose` yields an arbitrary element of the set in case it is not empty. This directly corresponds to Hilbert's ϵ-operator, written as "`SOME` $x.\ P(x)$" in Isabelle, denoting some x for which $P(x)$ is true, provided one exists [18].

Another interesting case to mention was the mapping of method `remove` that takes an object `elem` as argument. Isabelle's theory contains no operation that removes a single element from the set. Thus, `remove` had to be mapped to two other set operations: creation of a singleton set and set difference: `this - {elem}`.

Finally, we mention method `int_size`, which yields the number of elements the set contains. The method cannot be mapped to any term in the target theory since the theory does not define set cardinality. We discuss the consequences and solutions of such cases in Section 4.7.

An important issue of the mapping is the handling of equality. In general, we use reference equality for objects [3]. However, instances of model classes are treated as terms of a mathematical structure; therefore, the equality of this structure applies. We achieve this by overloading Isabelle's = operator. Instances of non-model classes are represented in Isabelle by a designated sort. The = operator on this sort denotes reference equality. Consequently, we simply map Java's == operator to Isabelle's = operator when applied to instances of non-model classes, in particular, to the elements stored in a `JMLObjectSet`. Instances of model classes are represented in Isabelle by the sort specified in the `mapped_to` clause of the model class. When applied to instances of model classes, we replace the == operator by a call to `equals`. This call is then mapped to Isabelle as prescribed by the `mapped_to` clause for `equals`. For instance, == operator on `JMLObjectSet` instances is mapped to set equality in Isabelle.

4.4 Consistency

We proved that the specifications of the model class are implied by the properties of Isabelle's `Set` theory. The proof was performed as described in Section 3.2.

We found one unsound equation in the equational theory. This equation intended to describe a relation between methods `remove` and `insert` as follows:

```
s.insert(e1).remove(e2).
  equals(e1 == e2 ? s : s.remove(e2).insert(e1))
```

where `s` is a `JMLObjectSet` instance, and `e1` and `e2` are two objects. The specification expresses that if `e1` and `e2` refer to the same object then inserting and removing the object in and from set `s` yields a set equivalent to `s`; otherwise, the order of performing the two operations is interchangeable.

Although this might look correct at first sight, the attempt to formally prove its correctness reveals that it is incorrect in case `s` contains `e1`, and `e1` and `e2` refer to the same object. In this case, the insertion yields some set s' that contains the same objects as `s` and the remove operation yields some set s'' that contains the same objects as s' except the object referenced by `e2` (and `e1`). Thus, this set cannot be equivalent to `s`.

This problem was directly pointed out by Isabelle via the open goal that remained after applying the automatic tactic **auto** on the corresponding lemma. The open goal was: $e2 : s \Rightarrow False$, expressing that the property does not hold in case `s` contains `e2`.

The buggy equation could be easily patched after the problem was caught and all specifications of the equational theory and the method specifications could be proven trivially using the **auto** tactic of Isabelle. As a consequence, we proved that the (patched) specifications of the model class are consistent.

4.5 Equivalence of equational theory and method specifications

While it was easy to notice the large overlap of properties specified by the class invariant and by the method specifications, it was not trivial to see whether they are equivalent. Thus, after having proved that the specifications are consistent, we proved their equivalence formally using Isabelle.

The procedure of proving the equivalence was the following. First, we declared signatures of function symbols the same way as described in Section 3.3. When proving that the equational theory implies the method specifications, we stated axioms based on the equational theory and generated lemmas based on the method specifications. Finally, we attempted to prove the lemmas by using the axioms. The other direction was proved analogously.

We found that the equational theory and the method specifications were not equivalent and none of them contained stronger specifications than the other. That is, while proving either direction, some lemmas could not be proven without strengthening some of the axioms or adding new ones. Four additional equations had to be added to the equational theory and one postcondition had to be strengthened in the method specifications in order to prove their equivalence. Here we give one example for each direction.

The equational theory contains two specifications that mention method `isEmpty`:

```
new JMLObjectSet().isEmpty()   and
!s.insert(e1).isEmpty()
```

These express that a newly allocated set is empty and that a set in which an element is inserted is not empty.
These specifications do not imply the property stated in the postcondition of method `isEmpty`:

```
\result == (\forall Object e; ; !this.has(e))
```

That is, `isEmpty` returns true if and only if the set does not contain any object. The postcondition could not be proven using the two equations because those just express *properties* of `isEmpty` (after construction and insertion) while the postcondition gives the *definition* of `isEmpty`.

Adding this definition to the equational theory (and thus to the set of axioms used in the proofs) solved the problem. In fact, the two original specifications could as well be removed since the new one (together with other properties) implies them.

The example where the method specifications had to be strengthened is the postcondition of `JMLObjectSet`'s constructor which takes an object `e` as argument and yields a set that contains `e`. The original postcondition "`this.has(e)`" was not sufficient to prove two specifications from the equational theory, for instance, the equation that relates the two constructors of the class:

```
new JMLObjectSet(e1).
  equals(new JMLObjectSet().insert(e1))
```

The weakness of the constructor's postcondition was again revealed by the open goal while proving the above equation and suggested us to strengthen the postcondition to express that object `e` is the one and only object contained by the set after construction:

```
(\forall Object e1; this.has(e1) <==> (e == e1))
```

The strengthened postcondition allowed us to prove the two remaining specifications in the equational theory.

To make sure that the added and strengthened specifications do not introduce unsoundness, we proved their consistency the same way as in Section 4.4.

The result of having proved the equivalence of the equational theory and the method specifications is that one can use either one or the other. For instance, one only needs to prove isomorphism of the method specifications and theory `HOL/Set` while the equational theory can be marked as redundant.

An interesting side-effect of this proof technique is that one can check whether specifications marked as redundant are indeed redundant. For instance, to check if a method specification marked as redundant is indeed implied by other method specifications, one needs to generate a lemma out of the specification marked as redundant and axioms from the non-redundant method specifications. If the lemma is provable, the specification is indeed redundant.

4.6 Completeness

As the last step we proved that the definitions of Isabelle's `Set` theory are implied by the (corrected and strengthened) specifications of `JMLObjectSet`. The proof was performed both for the equational theory and for method specifications and was carried out as described in Section 3.3. We note that due to the equivalence proof sketched above, it would have sufficed to perform this step either for the equational theory or for the method specifications. We carried out the proofs for both of them in order to gain more experience with our approach.

The most interesting part in this step was the mapping of Isabelle definitions to the signatures of the model class. Specifically, many of the definitions in Isabelle's `Set` theory use set comprehension. This is a construct that cannot be expressed by a method in the model class. However, probably for this reason, JML supports set comprehension on the syntax level [15]. The JML Reference Manual does not give a concrete definition for the semantics of the construct, thus we used the same meaning that Isabelle defines. This (1) ensured that we did not introduce unsoundness (provided the Isabelle definition is sound), and (2) gave a connection between mathematical set comprehension and the methods of `JMLObjectSet` since the Isabelle definition refers to set membership which corresponds to the `has` method of the model class.

With the help of set comprehension, most Isabelle definitions could be easily mapped back to the "language" of the model class. The corresponding lemmas could be proven both by the corrected and strengthened equational theory and by the strengthened method specifications. This means that both kinds of specifications are strong enough to imply the elementary properties of sets.

However, there were definitions that could not be mapped back to the model class in a straightforward way. An example is function *image* which takes a function *f* and a set *A* as parameters, and yields the image of set *A* under *f*. The model class does not provide such functionality and it cannot be expressed by the use of other methods of the class. Such cases lead us to the notion of *observational faithfulness*, discussed in the next session.

4.7 Mismatches between model class and structure

So far we only dealt with situations where each method of M had a direct correspondence in S and vice versa. However, this is not necessarily the case. If there is no direct correspondence, one can try to express the operation in terms of other operations (either of M or of S) that could already be mapped (directly or indirectly). As an example, in Section 4.3 we mentioned `JMLObjectSet`'s `remove` method, which could be expressed in terms of two functions of Isabelle's `Set`. In such cases the isomorphism result still holds.

However, there might be situations when no mapping exists and the operation cannot be expressed in terms of other ones. In such cases, there is a mismatch between M and S that cannot be bridged, that is, M and S are not isomorphic. However, the "direction" of the mismatch makes a difference in the consequences.

If a method of M cannot be translated to S then we cannot be sure that specifications referring to the method are indeed consistent and that the method semantically corresponds to some mathematical operation. An example for this situation is method `int_size` in `JMLObjectSet`. It has no counterpart in Isabelle's `HOL/Set` theory and cannot be expressed by other methods of the model class. This means that if we use theory `HOL/Set` we can neither guarantee consistency of specifications mentioning the method nor that the semantic meaning of the method is the intended one, namely set cardinality. In such situations, one needs to pick a different target theory where the mapping is possible. In our case Isabelle's `HOL/Finite_Set` could be picked as it provides function *card* to express set cardinality.

The situation is better if an operation of S cannot be translated to M. In this case the consistency of all methods in M can still be shown and the mappings prescribed by the `mapped_to` clauses can be safely used. That is, although isomorphism of M and S cannot be proven, isomorphism of all operations accessible in M and the corresponding operations in S can be shown. We call this kind of isomorphism *observational faithfulness* which is a sufficient result for the sound use of `mapped_to` clauses.

As mentioned above, `HOL/Set`'s function *image* cannot be mapped to `JMLObjectSet`. This means that the model class and the theory are not isomorphic. However, they are observationally faithful since isomorphism can still be shown for all methods of `JMLObjectSet` that may appear in specifications (apart from method `int_size`, as mentioned above).

5. Related work

The idea of using function symbols that are understood by the backend theorem prover directly on the specification level was already present in ESC/Java [9]. The special construct `\dttfsa` (*Damn The Torpedos, Full Speed Ahead!*) allowed users to refer to function applications on the level of Simplify, the theorem prover of ESC/Java. The corresponding function symbols were defined directly on the level of the prover. While this construct was a powerful means for specification, one had to be careful with its usage since on the specification level the definitions of the function symbols were hidden. The verification system did not give support for showing that the definitions were free of inconsistencies.

The Caduceus tool is a static verification system for C programs [7]. For specification and verification purposes the tool allows one to declare types and predicates as well as to define or axiomatize these predicates on the C source level. One can also define "hybrid" predicates, predicates that refer both to elements of the C program and elements of these specification-only types and predicates. Definitions of predicates can also be postponed on the source level and given directly in Coq, the backend prover of the tool. This concept eases the task of specifying and verifying programs since, for instance, it prevents the use of method calls in specifications and leads to definitions that are more suitable for provers than JML specifications. Case studies demonstrate the power of this approach [8, 11]. The drawback of the approach is the lack of consistency proof for definitions and axioms given on the source or prover level. This might lead to soundness issues.

Leavens et al. [14] identify the problem of specifying model types as a challenge for the specification and verification of programs. As a solution they propose the direct translation of model classes to mathematical theories, however, their proposal does not include details on how the translation would work and the issue of faithfulness is not mentioned.

Schoeller [20] roughly sketches the idea of the faithful mapping of model classes to mathematical structures. However, no details are given on how one would prove faithfulness.

Schoeller et al. developed a model library for Eiffel [21]. They address the faithfulness issue by equipping methods of model classes with specifications that directly correspond to axioms and theorems taken from mathematical textbooks. A shortcoming of this approach is that the resulting model library has to follow exactly the structure of the mimicked theory. This limits the design decisions one can make when composing the model library and it is unclear how one can support multiple theorem provers. Furthermore, user-defined model classes cannot be supported since there is no corresponding theory. Our approach allows more flexibility in the construction of model classes and libraries by using `mapped_to` clauses that can go beyond direct mappings since arbitrary terms of the target context can be specified. In turn, our approach requires one to prove faithfulness of the mapping.

Charles [3] proposes the introduction of the `native` keyword to JML in the context of work on the program verifier Jack [2]. The keyword can be attached to methods with a similar meaning to ESC/Java's `\dttfsa` construct: methods marked as `native` introduce uninterpreted function symbols and their definitions can be directly given on the level of Coq, the backend prover of Jack. Charles carries the idea over to model classes: the `native` keyword may also be attached to types with the meaning that such types get mapped to corresponding Coq datatypes. The mapping of `native` types is defined on the Coq level, too. This approach differs mainly in two ways from ours. First, our approach ensures faithfulness of the mappings. There is no attempt to do so in the work of Charles. Second, the `mapped_to` clause we propose in this paper allows one to specify the mappings on the specification language level. Furthermore, properties of model classes are specified in JML which typically provide easier understanding (for programmers) of the semantics than definitions given directly on the level of a theorem prover.

6. Conclusion

For the static verification of programs, model classes have to be mapped to mathematical structures of the underlying theorem prover. In this paper, we proposed an approach to show that this mapping is faithful by proving isomorphism between the model classes and the structures.

The proposed approach improves on previous work in three ways. First, previous work that proposed the direct translation of model-class methods to functions of a theorem prover does not ensure any actual semantic relationship between the mapped entities. This can easily lead to semantic mismatch between what was intended to be specified and what was actually verified.

Second, our approach leads to better specifications for model classes by ensuring their (relative) consistency and completeness. The identification and checking of redundant specifications further improves the quality of the specifications.

Third, previous work for ensuring the consistency of specifications of pure methods does not provide a satisfying solution in the presence of recursion [6]. The solution proposed by this paper solves this problem: by proving that a certain mathematical structure is a model for the specifications of a model class, we get the guarantee that the specifications are consistent. This result is independent of the presence of recursion.

To demonstrate our approach, we did a case study with a model class from JML's model library and a theory from Isabelle's library. The case study was successful in that observational faithfulness could be proved (except for method `int_size`) and interesting observations could be made on the model class: an incorrect specification was revealed, missing specifications were identified, and a precise relation between its equational theory and method specifications was identified.

Future work. Future work remains to provide tool support for the proposed mapping process described in this paper. Tools could support the typechecking of `mapped_to` clauses; the (partial) generation of proof scripts for faithfulness proofs; and the actual use of the mappings for static verification of programs.

For a better understanding of the strengths and weaknesses of our approach, further case studies with more complex model classes need to be done. In particular, it would be interesting to see how well our approach works for model classes that do not have a directly corresponding theory in the theorem prover, e.g., a stack.

Acknowledgments. We are grateful to Vijay d'Silva and Farhad Mehta for interesting discussions, and to the reviewers for helpful comments. Müller's work was done at ETH Zurich and was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-2005-015905 MOBIUS project.

References

[1] Y. Bertot and P. Castran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer, 2004.

[2] L. Burdy, A. Requet, and J.-L. Lanet. Java applet correctness: A developer-oriented approach. In *FME*, volume 2805 of *LNCS*, pages 422–439. Springer, 2003.

[3] J. Charles. Adding Native Specifications to JML. In *Formal Techniques for Java-like Programs*, 2006.

[4] Y. Cheon, G. Leavens, M. Sitaraman, and S. Edwards. Model variables: cleanly supporting abstraction in design by contract: Research articles. *Softw. Pract. Exper.*, 35(6):583–599, 2005.

[5] A. Darvas and K. R. M. Leino. Practical reasoning about invocations and implementations of pure methods. In *FASE*, volume 4422 of *LNCS*, pages 336–351. Springer, 2007.

[6] A. Darvas and P. Müller. Reasoning About Method Calls in Interface Specifications. *JOT*, 5(5):59–85, 2006.

[7] J.-C. Filliâtre, T. Hubert, and C. Marché. The Caduceus verification tool for C programs. Tutorial and Reference Manual. 2007.

[8] J.-C. Filliâtre and C. Marché. Multi-prover verification of C programs. In *ICFEM*, volume 3308 of *LNCS*, pages 15–29. Springer, 2004.

[9] C. Flanagan, K. R. M. Leino, M. Lillibridge, J. B. S. G. Nelson, and R. Stata. Extended static checking for Java. In *PLDI*, volume 37, pages 234–245. ACM Press, 2002.

[10] J. V. Guttag and J. J. Horning. *Larch: Languages and Tools for Formal Specification*. Texts and Monographs in Computer Science. Springer-Verlag, 1993.

[11] T. Hubert and C. Marché. A case study of C source code verification: the Schorr-Waite algorithm. In *SEFM*. IEEE Comp. Soc. Press, 2005.

[12] B. Jacobs and F. Piessens. Verification of programs with inspector methods. In *Formal Techniques for Java-like Programs*, 2006.

[13] G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. *Science of Computer Programming*, 55(1–3):185–205, 2005.

[14] G. T. Leavens, K. R. M. Leino, and P. Müller. Specification and verification challenges for sequential object-oriented programs. *Formal Aspects of Computing*, 2007. To appear.

[15] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, and J. Kiniry. *JML Reference Manual*. Iowa State University, Last revised February 2007.

[16] J. Meyer, P. Müller, and A. Poetzsch-Heffter. The JIVE system—implementation description. 2000.

[17] M. Miragliotta. Specification model library for the interactive program prover JIVE. ETH Zurich, Semester Thesis, 2004.

[18] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[19] T. Nipkow, L. C. Paulson, and M. Wenzel. Theory HOL/Set from "The Isabelle Library". `isabelle.in.tum.de/library/HOL/Set.html`, 2005.

[20] B. Schoeller. Strengthening Eiffel contracts using models. In *Formal Aspects of Component Software*, 2003.

[21] B. Schoeller, T. Widmer, and B. Meyer. Making specifications complete through models. In *Architecting Systems with Trustworthy Components*, volume 3938 of *LNCS*. Springer, 2006.

[22] N. Shankar, S. Owre, and J. M. Rushby. A Tutorial on Specification and Verification Using PVS (Beta Release). Technical report, Computer Science Laboratory, SRI International, March 1993.

Proof-Transforming Compilation of Programs with Abrupt Termination

Peter Müller
Microsoft Research, USA
mueller@microsoft.com

Martin Nordio
ETH Zurich, Switzerland
Martin.Nordio@inf.ethz.ch

ABSTRACT

The execution of untrusted bytecode programs can produce undesired behavior. A proof on the bytecode programs can be generated to ensure safe execution. Automatic techniques to generate proofs, such as certifying compilation, can only be used for a restricted set of properties such as type safety. Interactive verification of bytecode is difficult due to its unstructured control flow. Our approach is verify programs on the source level and then translate the proof to the bytecode level. This translation is non-trivial for programs with abrupt termination. We present proof transforming compilation from Java to Java Bytecode. This paper formalizes the proof transformation and present a soundness result.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Program Verification—*Correctness proofs*; D.3.4 [**Programming Languages**]: Processors—*Compilers*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs

General Terms

Verification, Languages

Keywords

Trusted Components, Proof-Carrying Code, Proof-Transforming Compiler

1. INTRODUCTION

Proof-Carrying Code (PCC) [8, 9] has been developed with the goal of solving the problems produced by the unsafe execution of mobile code. In PCC, the code producer provides a *proof*, a certificate that the code does not violate the security properties of the code consumer. Before the code execution, the proof is checked by the code consumer. Only if the proof is correct, the code is executed.

Sixth International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2007), September 3-4, 2007, Cavtat near Dubrovnik, Croatia.

The certificate proves the properties that are satisfied by the bytecode program. With the goal of generating certificates automatically, Necula [9] has developed certifying compilers. *Certifying compilers* are compilers that take a program as input and produce bytecode and its proof. Unfortunately, certifying compilers only work with a restricted set of provable properties such as type safety.

Another approach to solve the problem caused by mobile code is *interactive verification of bytecode*. This approach is applicable to a wide range of properties, but is difficult due to the bytecode's unstructured control flow. Contrary, source verification is simpler, but does not generate a certificate for the bytecode program.

The approach we propose here is the use of a Proof - Transforming Compiler (PTC). PTCs are similar to certifying compilers in PCC, but take a source proof as input and produce the bytecode proof. Figure 1 shows the architecture of this approach. The code producer develops a program. A proof of the source program is developed using a prover. Then, the PTC translates the proof producing the bytecode and its proof, which are sent to the code consumer. The proof checker verifies the proof. If the source proof or the translation were incorrect, the checker would reject the code.

An important property of Proof-Transforming Compilers is that they do not have to be trusted. If the compiler produces a wrong specification or a wrong proof for a component, the proof checker will reject the component. This approach has the strengths of both above mentioned approaches.

If the source and target languages are close, the proof translation is simple. However, if they are not close and the compilation function is complex, the translation can be hard. For example, proof-transformation from a subset of Java with `try-catch`, `try-finally` and `break` statements to Java Bytecode is not simple. Compiling these statements in isolation is simple, but the compilation of their interplay is not.

A `try-finally` statement is compiled using *code duplication*: the `finally` block is put after the `try` block. If `try-finally` statements are used inside of a `while` loop, the compilation of `break` statements first duplicates the `finally` blocks and then inserts a jump to the end of the loop. Furthermore, the generation of exception tables is also harder. The code duplicated before the `break` may have exception handlers different from those of the enclosing `try` block. Therefore, the exception table must be changed so that exceptions are caught by the appropriate handlers. In this paper, we present the first PTC that handles these complications.

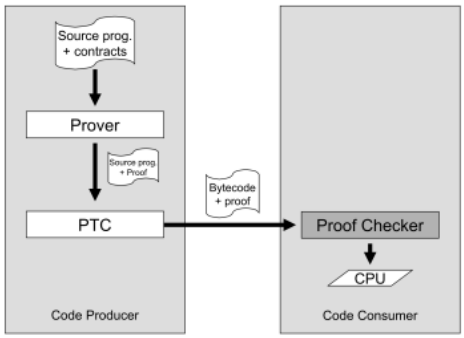


Figure 1: General architecture.

Outline. The source language and its Hoare-style logic are introduced in Section 2. We present the Bytecode language and its logic in Section 3. In Section 4, we define the proof transformation. Section 5 illustrates proof transformations by an example. Section 6 states a soundness theorem. Related work is discussed in Section 7. Section 8 summarizes and gives directions for future work.

2. SOURCE LANGUAGE AND LOGIC

The source language we consider is similar to a Java subset. Its definition is the following:

$$
\begin{array}{lcl}
\mathit{exp} & ::= & \mathit{literal} \mid \mathit{var} \mid \mathit{exp}\ \mathit{op}\ \mathit{exp} \\
\mathit{stm} & ::= & x = \mathit{exp} \mid \mathit{stm}; \mathit{stm} \mid \texttt{while}\ (\mathit{exp})\ \mathit{stm} \\
 & \mid & \texttt{break}\ ; \mid \texttt{if}\ (\mathit{exp})\ \mathit{stm}\ \texttt{else}\ \mathit{stm} \\
 & \mid & \texttt{try}\ \mathit{stm}\ \texttt{catch}\ (\mathit{type}\ \mathit{var})\ \mathit{stm} \\
 & \mid & \texttt{try}\ \mathit{stm}\ \texttt{finally}\ \mathit{stm} \mid \texttt{throw}\ \mathit{exp}\ ;
\end{array}
$$

To avoid `return` statements, we assume that the return value of every method is assigned to a special local variable named **result** (this is the only discordance with respect to Java). Moreover, we assume that the expressions are side-effect-free and cannot throw exceptions.

The subset of Java is small, but the combination of `while`, `break`s, `try-catch` and `try-finally` statements produces an interesting subset especially from the point of view of compilation. The code duplication used by the compiler for `try-finally` statements increases the complexity of the compilation and translation functions, specially the formalization and its soundness proof.

In our technical report [7], the source languages also includes object-oriented features such as cast, new, read and write field, and method invocation. In this paper, we only present the most interesting features.

2.1 Method and statement specifications

The logic is based on the programming logic introduced in [6, 12, 13]. We have modified it and proposed new rules for `while` including `break` and exceptions, `try-catch` and `try-finally`. In [13], a special variable χ is used to capture the status of the program such as normal or exceptional status. This variable is not necessary in the bytecode proof since non-linear control flow is implemented via jumps. To eliminate the χ variable, we use Hoare triples with two or three postconditions to encode the status of the program execution. This simplifies not only the translation but also the presentation.

Properties of methods are expressed by Hoare triples of the form $\{P\}\ \ T.m\ \ \{\ Q_n\ ,\ Q_e\ \}$, where P, Q_n, Q_e are first-order formulas and $T.m$ is a method m declared in class T. The third component of the triple consists of a normal postcondition (Q_n), and an exceptional postcondition (Q_e). We call such a triple *method specification*.

Properties of statements are specified by Hoare triples of the form $\{P\}\ \ S\ \ \{Q_n, Q_b, Q_e\}$, where P, Q_n, Q_b, Q_e are first-order formulas and S is a statement. For statements, we have a normal postcondition (Q_n), a postcondition after the execution of a `break` (Q_b), and an exceptional postcondition (Q_e).

The triple $\{P\}\ \ S\ \ \{Q_n, Q_b, Q_e\}$ defines the following refined partial correctness property: if S's execution starts in a state satisfying P, then (1) S terminates normally in a state where Q_n holds, or S executes a `break` statement and Q_b holds, or S throws an exception and Q_e holds, or (2) S aborts due to errors or actions that are beyond the semantics of the programming language, e.g., memory allocation problems, or (3) S runs forever.

2.2 Rules

Figure 2 shows the rules for compositional, `while`, `break`, `try-catch`, and `throw` statements. In the compositional statement, the statement s_1 is executed first. The statement s_2 is executed if and only if s_1 has terminated normally.

In the `while` rule, the execution of the statement s_1 can produce three results: either (1) s_1 terminates normally and I holds, or (2) s_1 executes a `break` statement and Q_b holds, or (3) s_1 throws an exception and R_e holds. The postcondition of the `while` statement expresses that either the loop terminates normally and $(I \wedge \neg e) \vee\ \ Q_b$ holds or throws an exception and R_e holds. The `break` postcondition is false, because after a `break` within the loop, execution continues normally after the loop.

The `break` rule sets the normal and exception postcondition to false and the `break` postcondition to P due to the execution of a `break` statement.

In the `try-catch` rule, the execution of the statement s_1 can produce three different results: (1) s_1 terminates normally and Q_n holds or terminates with a `break` and Q_b holds. In these cases, the statement s_2 is not executed and the postcondition of the `try-catch` is the postcondition of s_1; (2) s_1 throws an exception and the exception is not caught. The statement s_2 is not executed and the `try-catch` finishes in an exception mode. The postcondition is $Q_e'' \wedge \tau(excV) \not\preceq T$, where τ yields the runtime type of an object, $excV$ is a variable that stores the current exception, and $\preceq$ denotes subtyping; (3) s_1 throws an exception and the exception is caught. In the postcondition of s_1, $Q_e' \ \wedge\ \tau(excV) \preceq T$ specifies that the exception is caught. Finally, s_2 is executed producing the postcondition. Note that the postcondition is not only a normal postcondition: it also has to take into account that s_2 can throw an exception or can execute a `break`.

Similar to `break`, the `throw` rule modifies the postcondition P by updating the exception component of the state with the just evaluated reference.

To define the rule for `try-finally`, we have to treat a special case, illustrated through the example in Figure 3.

The exception thrown in the `try` block is never caught. However, the loop terminates normally due to the execution

compositional

$$\frac{\{P\}\ \ s_1\ \ \{Q_n, R_b, R_e\} \quad \{Q_n\}\ \ s_2\ \ \{R_n, R_b, R_e\}}{\{P\}\ \ s_1; s_2\ \ \{R_n, R_b, R_e\}}$$

while

$$\frac{\{e\ \wedge\ I\}\ \ s_1\ \ \{I, Q_b, R_e\}}{\{I\}\ \ \texttt{while}\ (e)\ s_1\ \ \{((I \wedge \neg e) \vee\ Q_b), \mathit{false}, R_e\}}$$

break

$$\frac{}{\{P\}\ \ \texttt{break}\ \ \{\mathit{false}, P, \mathit{false}\}}$$

try-catch

$$\frac{\{P\}\ \ s_1\ \ \{Q_n, Q_b, Q\} \quad \{Q'_e[e/excV]\}\ \ s_2\ \ \{Q_n, Q_b, R_e\}}{\{P\}\ \ \texttt{try}\ s_1\ \texttt{catch}\ (T\ e)\ s_2\ \ \{Q_n, Q_b, R\}}$$

where

$Q \equiv (\ (Q''_e\ \wedge\ \tau(excV) \not\preceq T) \vee (Q'_e\ \wedge\ \tau(excV) \preceq T)\ )$

$R \equiv (R_e\ \vee\ (Q''_e\ \wedge\ \tau(excV) \not\preceq T)\ )$

throw

$$\frac{}{\{P[e/excV]\}\ \ \texttt{throw}\ e\ \ \{\mathit{false}, \mathit{false}, P\}}$$

Figure 2: Rules for composition, `while`, `break`, `try-catch`, and `throw`.

```
void foo () {
  int b = 1;
  while (true) {
    try { throw new Exception(); }
    finally { b++; break; }
  }
  b++;
}
```

Figure 3: The exception raised in the `try` block is not handled, yet the method terminates normally.

of the `break` statement in the `finally` block. Thus, the value of b at the end of *foo* is 3.

If an exception occurs in a `try` block, it will be re-raised after the execution of the `finally` block. If both the `try` and the `finally` block throw an exception, the latter takes precedence. The following table summarizes the status of the program after the execution of the `try-finally`:

		finally		
		normal	**break**	**exc$_2$**
try	**normal**	normal	break	exc_2
	break	break	break	exc_2
	exc$_1$	exc_1	break	exc_2

We use the fresh variable *eTmp* to store the exception occurred in s_1 because another exception might be raised and caught in s_2. In this case, we still need to have access to the first exception of s_1 because this exception is the result of that statement [13]. We use the fresh variable $\mathcal{X}\,Tmp$ to store the status of the program after the execution of s_1. The possible values of $\mathcal{X}\,Tmp$ are: *normal*, *break*, and *exc*. Depending on the status after the execution of s_2, we need to propagate an exception or change the status of the program to *break*. The rule is the following:

$$\frac{\{P\}\ \ s_1\ \ \{Q_n, Q_b, Q_e\} \quad \{Q\}\ \ s_2\ \ \{R, R'_b, R'_e\}}{\{P\}\ \ \texttt{try}\ s_1\ \texttt{finally}\ s_2\ \ \{R'_n, R'_b, R'_e\}}$$

where

$$Q \equiv \left(\begin{array}{l}(Q_n \wedge \mathcal{X}\,Tmp = \mathit{normal})\ \vee (Q_b \wedge \mathcal{X}\,Tmp = \mathit{break}) \vee \\ (\ Q_e[eTmp/excV] \wedge \mathcal{X}\,Tmp = exc \wedge eTmp = excV\ )\end{array}\right)$$

$$R \equiv \left(\begin{array}{l}(R'_n \wedge \mathcal{X}\,Tmp = \mathit{normal})\ \vee\ (R'_b \wedge \mathcal{X}\,Tmp = \mathit{break}) \vee \\ (R'_e \wedge \mathcal{X}\,Tmp = exc)\end{array}\right)$$

Furthermore, the logic contains language-independent rules such as the rule of consequence. Due to space limitations, we do not present them here.

3. BYTECODE LANGUAGE AND LOGIC

The bytecode language consists of classes with fields and methods. Methods are implemented as method bodies consisting of a sequence of labeled bytecode instructions. Bytecode instructions operate on the operand stack, local variables (which also include parameters), and heap. The bytecode instructions used to compile the source language are: pushc v, pushv x, pop x, bin_{op}, goto l, brtrue l, and athrow. pushc v pushes constant v onto the stack. pushv x pushes the value of a variable x onto the stack. pop x pops the topmost element off the stack and assigns it to the local variable x. bin_{op} removes the two topmost values from the stack and pushes the result of applying bin_{op} to these values. goto l transfers control to the point l. brtrue l transfers control to the point l if the topmost element of the stack is true and unconditionally pops it. athrow takes the topmost value from the stack, assumed to be an exception, and throws it. To simplify the translation of source programs, we assume the bytecode language has a type boolean.

The bytecode logic is a Hoare-style program logic which allows one to formally verify that implementations satisfy interface specifications given as pre- and postconditions. We use the bytecode logic developed by Bannwart and Müller [1].

3.1 Method and Instruction Specifications

To make proof transformation feasible, it is essential that the source logic and the bytecode logic are similar in their structure. In particular, they treat methods in the same way, they contain the same language-independent rules, and triples have a similar meaning.

Analogously to the source logic, properties of methods are expressed by method specifications of the form form {P} *T.mp* $\{Q_n,\ Q_e\}$. Properties of method bodies are expressed by Hoare triples of the form {P} *comp* {Q}, where P, Q are first-order formulas and *comp* is a method body. The triple {P} *comp* {Q} expresses the following refined partial correctness property: if the execution of *comp* starts in a state satisfying P, then (1) *comp* terminates in a state where Q holds, or (2) *comp* aborts due to errors or actions that are beyond the semantics of the programming language, or (3) *comp* runs forever.

The unstructured control flow of bytecode programs makes it difficult to handle instruction sequences, because jumps can transfer control into and from the middle of a sequence. Therefore, the logic treats each instruction individually: each individual instruction I_l in a method body p has a precondition E_l. An instruction with its precondition is called an *instruction specification*, written as $\{E_l\}\ l : I_l$.

The meaning of an instruction specification $\{E_l\}\ l : I_l$ cannot be defined in isolation. $\{E_l\}\ l : I_l$ expresses that if the precondition E_l holds when the program counter is at position l, the precondition $E_{l'}$ of I_l's successor instruction I'_l holds after normal termination of I_l.

3.2 Rules

All the rules for instructions, except for method calls, have the following form:

$$\frac{E_l \Rightarrow wp_p^1(I_l)}{\mathrm{A} \vdash \{E_l\}\ l : I_l}$$

where $wp_p^1(I_l)$ denotes the *local weakest precondition* of instruction I_l. Such a rule specifies that the precondition of I_l has to imply the weakest precondition of I_l with respect to all possible successor instructions of I_l. The definition of wp_p^1 is shown in Table 1.

Within an assertion, the current stack is referred to as s and its elements are denoted by non-negative integers: element 0 is the topmost element, etc. The interpretation $[E_l] : \mathit{State} \times \mathit{Stack} \rightarrow \mathit{Value}$ for s is

$$\begin{array}{l}[s(0)]\langle S, (\sigma, v)\rangle \ = v \ \text{ and} \\ [s(i+1)]\langle S, (\sigma, v)\rangle = [s(i)]\langle S, \sigma\rangle\end{array}$$

The functions *shift* and *unshift* define the substitutions that occur when values are pushed onto and popped from the stack, respectively:

$$\begin{array}{ll}\mathit{shift}(E) & = E[s(i+1)/s(i) \mid \forall i \in \mathbb{N}] \\ \mathit{unshift} & = \mathit{shift}^{-1}\end{array}$$

I_l	$wp_p^1(I_l)$
pushc v	$\mathit{unshift}(E_{l+1}[v/s(0)])$
pushv x	$\mathit{unshift}(E_{l+1}[x/s(0)])$
pop x	$(\mathit{shift}(E_{l+1}))[s(0)/x]$
bin_{op}	$(\mathit{shift}(E_{l+1}))[s(1)\ op\ s(0)/s(1)]$
goto l'	$E_{l'}$
brtrue l'	$(\neg s(0) \Rightarrow \mathit{shift}(E_{l+1})) \wedge (s(0) \Rightarrow \mathit{shift}(E_{l'}))$

Table 1: Definition of function wp_p^1.

4. PROOF TRANSLATION

Our proof-transforming compiler is based on two *transformation functions*, ∇_S and ∇_E, for statements and expressions, respectively. Both functions yield a sequence of bytecode instructions and their specification. The PTC takes a list of classes with their proofs and returns the bytecode classes with their proofs.

The function ∇_E generates a bytecode proof from a source expression and a precondition for its evaluation. The function ∇_S generates a bytecode proof and an exception table from a source proof. These functions are defined as a composition of the translations of its sub-trees. The signatures are the following:

$$\begin{array}{ll}\nabla_E & : \mathit{Precondition} \times \mathit{Expression} \times \mathit{Postcondition} \times \\ & \quad \mathit{Label} \rightarrow \mathit{BytecodeProof}\end{array}$$

$$\begin{array}{ll}\nabla_S & : \mathit{ProofTree} \times \mathit{Label} \times \mathit{Label} \times \mathit{Label} \times \mathit{List}[\mathit{Finally}] \times \\ & \quad \mathit{ExcTable} \rightarrow [\mathit{BytecodeProof} \times \mathit{ExcTable}]\end{array}$$

In ∇_E, the label is used as the starting label of the translation. *ProofTree* is a derivation in the source logic. In ∇_S, the three labels are: (1) l_{start} for the first label of the resulting bytecode; (2) l_{next} for the label after the resulting bytecode; this is for instance used in the translation of an `else` branch to determine where to jump at the end; (3) l_{break} for the jump target for `break` statements.

Type	Typical use
$\mathit{Precondition} \cup \mathit{Postcondition}$	P, Q, R, U, V
ProofTree (for source language only)	$T_{S_1}, T_{S_2}, \mathit{Tree}_i$
ProofTree (for `finally` only)	T_{F_i}
$\mathit{List}[\mathit{Finally}]$	f
ExceptionTable	et_i
ExceptionTable (for `finally` only)	et_i'
BytecodeProof	B_{S_1}, B_{S_2}
InstrSpec	$b_{\mathsf{pushc}}, \ldots, b_{\mathsf{brtrue}}$
Label	$l_{start}, l_{next}, l_{break}, l_b, l_c, \ldots, l_g$

Table 2: Naming conventions.

The *BytecodeProof* type is defined as a list of *InstrSpec*, where *InstrSpec* is an instruction specification. The *Finally* type, used to translate `finally` statements, is defined as a tuple $[\mathit{ProofTree}, \mathit{ExcTable}]$. Furthermore, the ∇_S takes an exception table as parameter and produces an exception table. This is necessary because the translation of `break` statements can lead to a modification of the exception table as described above. (more details are presented in Section 4.3).

The *ExcTable* type is defined as follows:

$$\begin{array}{ll}\mathit{ExcTable} & := \mathit{List}[\mathit{ExcTableEntry}] \\ \mathit{ExcTableEntry} & := [\mathit{Label}, \mathit{Label}, \mathit{Label}, \mathit{Type}]\end{array}$$

In the *ExcTableEntry* type, the first label is the *starting label* of the exception line, the second denotes the *ending label*, and the third is the *target label*. An exception of type T_1 thrown at line l is caught by the exception entry $[l_{start}, l_{end}, l_{targ}, T_2]$ if and only if $l_{start} \leq l < l_{end}$ and $T_1 \preceq T_2$. Control is then transferred to l_{targ}.

In the following, we present the proof translation for compositional rule, `while`, `try-finally`, and `break`. Table 2 comprises the naming conventions we use in the rest of this paper.

4.1 Compositional Statement

Let T_{S_1} and T_{S_2} be the following proof trees:

$$T_{S_1} \equiv \frac{\mathit{Tree}_1}{\{P\}\ \ s_1\ \ \{Q_n, R_b, R_e\}}$$

$$T_{S_2} \equiv \frac{\mathit{Tree}_2}{\{Q_n\}\ \ s_2\ \ \{R_n, R_b, R_e\}}$$

$$T_{S1;S2} \equiv \frac{T_{S_1} \qquad T_{S_2}}{\{P\}\ \ s_1; s_2\ \ \{R_n, R_b, R_e\}}$$

In the translation of T_{S_1}, the label l_{next} is the start label of the translation of s_2, say l_b. The translation of T_{S_2} uses the exception table produced by the translation of T_{S_1}, et_1. The translation of $T_{S1;S2}$ yields the concatenation of the bytecode proofs for the sub-statements and the exception table produced by the translation of T_{S_2}.

Let $[B_{S_1}, et_1]$ and $[B_{S_2}, et_2]$ be of type $[\mathit{BytecodeProof}, \mathit{ExcTable}]$:

$$\begin{array}{l}[B_{S_1}, et_1] = \nabla_S\left(T_{S_1}, l_{start}, l_b, l_{break}, f, et\right) \\ [B_{S_2}, et_2] = \nabla_S\left(T_{S_2}, l_b, l_{next}, l_{break}, f, et_1\right)\end{array}$$

The translation is defined as follows:

$$\nabla_S\ (\ T_{S1;S2},\ l_{start}, l_{next}, l_{break}, f,\ et) = \\ [B_{S_1} + B_{S_2}\ ,\ et_2]$$

The bytecode for s_1 establishes Q_n, which is the precondition of the first instruction of the bytecode for s_2. Therefore, the concatenation $B_{S_1} + B_{S_2}$ produces a sequence of valid instruction specifications. We will formalize soundness in Section 6.

4.2 While Statement

Let T_{S_1} and T_{while} be the following proof trees:

$$T_{S_1} \equiv \frac{Tree_1}{\{e\ \wedge\ I\}\ \ s_1\ \ \{I, Q_b, R_e\}}$$

$$T_{while} \equiv \frac{T_{S_1}}{\{I\}\ \ \texttt{while}\ (e)\ s_1\ \ \{(I \wedge \neg e) \vee\ \ Q_b, false, R_e\}}$$

In this translation, first the loop expression is evaluated at l_c. If it is true, control is transferred to l_b, the start label of the loop body. In the translation of T_{S_1}, the start label and next labels are l_b and l_c. The break label is the end of the loop (l_{next}). Furthermore, the finally list is set to $\emptyset$, because a `break` inside the loop jumps to the end of the loop without executing any `finally` blocks.

Let b_{goto} and b_{brtrue} be instruction specifications and B_{S_1} and B_e be bytecode proofs:

$$\begin{aligned} b_{\mathsf{goto}} = &\quad \{I\}\quad l_a : \mathsf{goto}\ l_c \\ [B_{S_1}, et_1] = &\quad \nabla_S\,(T_{S_1},\ l_b, l_c, l_{next},\ \emptyset, et) \\ B_e = &\quad \nabla_E\ (\ I,\ e,\ (shift(I)\ \wedge\ s(0) = e)\ ,_c\ ) \\ b_{\mathsf{brtrue}} = &\quad \{shift(I)\ \wedge\ s(0) = e\}\quad l_d :\ \mathsf{brtrue}\ l_b \end{aligned}$$

The definition of the translation is the following:

$$\nabla_S\,(T_{while},\ l_{start}, l_{next}, l_{break},\ f,\ et) = \\ [\ b_{\mathsf{goto}} + B_{S_1} + B_e + b_{\mathsf{brtrue}}\ ,\ et_1\ ]$$

The instruction b_{goto} establishes I, which is the precondition of the successor instruction (the first instruction of B_e). B_e establishes $shift(I)\ \wedge\ s(0) = e$ because the evaluation of the expression pushes the result on top of the stack. This postcondition implies the precondition of the successor instruction b_{brtrue}. b_{brtrue} establishes the preconditions of both possible successor instructions, namely $e\ \wedge\ I$ for the successor l_b (the first instruction of B_{S_1}), and $I\ \wedge\ \neg e$ for l_{next}. Finally, B_{S_1} establishes I, which implies the precondition of its successor B_e, I. Therefore, the produced bytecode proof is valid.

4.3 Try-Finally Statement

Sun's newer Java compilers translate `try-finally` statements using code duplication. Consider the following example:

```
while (i < 20) {
    try {
        try {
            try { ... break; ... }
            catch (Exception e) { i = 9; }
        }
        finally { throw new Exception(); }
    }
    catch (Exception e) { i = 99; }
}
```

The `finally` body is duplicated before the `break`. But the exception thrown in the `finally` bock must be caught by the outer `try-catch`. To achieve that, the compiler creates, in the following order, exception lines for the outer `try-catch`, for the `try-finally`, and for the inner `try-catch`. When the compiler reaches the `break`, it divides the exception entry of the inner `try-catch` and `try-finally` into two parts so that the exception is caught by the outer `try-finally`. To be able to divide the exception table the compiler needs to compare the exception entries. This is why our *Finally* type consists of a proof tree (for the duplicated code) and an exception table. Note that we have a list of *Finally* to handle nested `try-finally` statements.

Let T_{S_1}, T_{S_2} and $T_{try-finally}$ be the following proof trees:

$$T_{S_1} \equiv \frac{Tree_1}{\{P\}\ \ s_1\ \ \{Q_n, Q_b, Q_e\}}$$

$$T_{S_2} \equiv \frac{Tree_2}{\{Q\}\ \ s_2\ \ \{R, R'_b, R'_e\}}$$

$$T_{try-finally} \equiv \frac{T_{S_1} \qquad T_{S_2}}{\{P\}\ \ \texttt{try}\ s_1\ \mathsf{finally}\ s_2\ \ \{R'_n, R'_b, R'_e\}}$$

where

$$Q \equiv \left(\begin{array}{l} (Q_n \wedge \mathcal{X}\,Tmp = normal)\ \vee (Q_b \wedge \mathcal{X}\,Tmp = break) \vee \\ (\ \ Q_e[eTmp/excV] \wedge \mathcal{X}\,Tmp = exc \wedge eTmp = excV\ \ ) \end{array}\right)$$

$$R \equiv \left(\begin{array}{l} (R'_n \wedge \mathcal{X}\,Tmp = normal)\ \vee\ (R'_b \wedge \mathcal{X}\,Tmp = break) \vee \\ (R'_e \wedge \mathcal{X}\,Tmp = exc) \end{array}\right)$$

In this translation, the bytecode for s_1 is followed by the bytecode for s_2. In the translation of T_{S_1}, the `finally` block is added to the finally-list f with T_{S_2}'s source proof tree and its associated exception table. The corresponding exception table is retrieved using the function $getExcLines$: $Label \times Label \times ExcTable \rightarrow ExcTable$. Given two labels and an exception table et, $getExcLines$ returns, per every exception type in et, the first et's exception entry (if any) for which the interval made by the starting and ending labels includes the two given labels. Furthermore, a new exception entry, for the `finally` block, is added to the exception table et. Then, the bytecode proof for the case when s_1 throws an exception is created. The exception table of this translation is produced by the predecessor translations.

Let et', et'' be the following exception tables:

$$et_1 = et + [l_{start}, l_b, l_d, any] \\ et' = getExcLines(l_a, l_b, et_1)$$

Let b_{goto}, b_{pop}, b_{pushv}, and b_{athrow} be instructions specifications and B_{S_1}, B_{S_2}, and B'_{S_2} be bytecode proofs:

$$\begin{aligned} [B_{S_1}, et_2] = &\quad \nabla_S\ \ (T_{S_1},\ l_{start},\ l_b,\ l_{break},\ [T_{S_2}\ , et'] + f,\ \ et_1) \\ [B_{S_2}, et_3] = &\quad \nabla_S\ \ (T_{S_2},\ l_b,\ \ l_c,\ \ l_{break},\ f,\ \ et_2) \\ b_{\mathsf{goto}} = &\quad \{Q'_n\} \qquad\qquad\qquad\qquad l_c : \mathsf{goto}\ l_{next} \\ b_{\mathsf{pop}} = &\quad \left\{\begin{array}{l} shift(Q_e) \wedge \\ excV \neq null \\ \wedge\ s(0) = excV \end{array}\right\} \qquad l_d : \mathsf{pop}\ eTmp \\ [B_{S'_2}, et_4] = &\quad \nabla_S\ \ (T_{S_2}, l_e, l_f, l_{break},\ f,\ \ et_3) \\ b_{\mathsf{pushv}} = &\quad \left\{\ \ Q'_n\ \ \vee\ \ Q'_b\ \ \vee\ \ Q'_e\ \ \right\} \qquad l_f : \mathsf{pushv}\ eTmp \\ b_{\mathsf{athrow}} = &\quad \left\{\begin{array}{l} (Q'_n\ \ \vee\ \ Q'_b\ \ \vee\ \ Q'_e) \\ \wedge\ s(0) = eTmp \end{array}\right\} \qquad l_g : \mathsf{athrow} \end{aligned}$$

The translation is defined as follows:

$$\nabla_S\ (\ T_{try-finally},\ l_{start}, l_{next}, l_{break},\ f,\ et) = \\ [\ B_{S_1} + B_{S_2} + b_{\mathsf{goto}} + b_{\mathsf{pop}} + B_{S_2'} + b_{\mathsf{pushv}} + b_{\mathsf{athrow}}\ ,\ et_4\ ]$$

It is easy to see that the instruction specifications b_{goto}, b_{pop}, b_{pushv}, and b_{athrow} are valid (by applying the definition of the weakest precondition). However, the argument for the translation of T_{S_1} and T_{S_2} is more complex. Basically, the result is a valid proof because the proof tree inserted in f for the translation of T_{S_1} is a valid proof and the postcondition of each finally block implies the precondition of the next one. Furthermore, for normal execution, the postcondition of B_{S_1} (Q_n) implies the precondition of B_{S_2} (Q).

4.4 Break Statement

To specify the rules for `break`, we use the following recursive function: *divide*: *ExcTable* × *ExcTableEntry* × *Label* × *Label* → *ExcTable*. Its definition assumes that the exception entry is in the given exception table and the two given labels are in the interval made by the exception entry's starting and ending labels. Given an exception entry y and two labels l_s and l_e, *divide* compares every exception entry, say x, of the given exception table to y. If the interval defined by x's starting and ending labels is included in the interval defined by y's starting and ending labels, then x must be divided to have the appropriate behavior of the exceptions. Thus, the first and the last interval of the three intervals defined by x's starting and ending labels, l_s, and l_e are returned, and the procedure is continued for the next exception entry. If x and y are equal, then recursion stops as *divide* reached the expected entry. The formal definition of *divide* is the following:

$$\begin{array}{l} \mathit{divide} : \mathit{ExcTable} \times \mathit{ExcTableEntry}\ \times \mathit{Label} \times \\ \qquad \mathit{Label} \rightarrow \mathit{ExcTable} \\ \mathit{divide} : ([\,], e', l_s, l_e) = [\ e'\ ] \\ \mathit{divide} : (e : et, e', l_s, l_e) = \\ \quad [\ l_{start},\ l_s,\ l_{targ},\ T_1\ ] + [\ l_e,\ l_{end},\ l_{targ},\ T_1\ ] + \\ \qquad \mathit{divide}(et, e', l_s, l_e) \quad \textbf{if}\ e \subseteq e'\ \wedge\ e \neq e' \\ \quad \mid e : et \quad \textbf{if}\ e = e' \\ \quad \mid e : \mathit{divide}(et, e', l_s, l_e) \quad \textbf{otherwise} \end{array}$$

where

$e \equiv [l_{start}, l_{end}, l_{targ}, T_1]$ and $e' \equiv [l'_{start}, l'_{end}, l'_{targ}, T_2]$

$$\begin{array}{l} \subseteq : \mathit{ExcTableEntry} \times \mathit{ExcTableEntry} \rightarrow \mathit{Boolean} \\ \subseteq : ([l_{start}, l_{end}, l_{targ}, T_1], [l'_{start}, l'_{end}, l'_{targ}, T_2]) = \\ \quad \mathit{true} \quad \textbf{if}\ (l'_{st} \leq l_{st})\ \wedge\ (l'_{end} \geq l_{end}) \\ \quad \mid \mathit{false}\ \textbf{otherwise} \end{array}$$

When a `break` statement is encountered, the proof tree of every `finally` block the `break` has to execute upon exiting the loop is translated. Then, control is transferred to the end of the loop using the label l_{break}. Let $f_i = [T_{F_i}, et'_i]$ denote the i-th element of the list f, where

$$T_{F_i} = \frac{\mathit{Tree}_i}{\{\,U^i\,\}\ \ s_i\ \ \{\,V^i\,\}}$$

and U^i and V^i have the following form, which corresponds to the Hoare rule for `try-finally` (see Section 2):

$$U^i \equiv \left\{ \begin{array}{l} (U^i_n\ \wedge\ \mathcal{X}\mathit{Tmp} = \mathit{normal}) \vee \\ (U^i_b\ \wedge\ \mathcal{X}\mathit{Tmp} = \mathit{break}) \vee \\ \left(\begin{array}{l} U^i_e[\mathit{eTmp}/\mathit{excV}]\ \wedge\ \mathcal{X}\mathit{Tmp} = \mathit{exc}\ \wedge \\ \mathit{eTmp} = \mathit{excV} \end{array}\right) \end{array} \right\}$$

$$V^i \equiv \left\{ \begin{array}{l} \left(\begin{array}{l} (V'^i_n\ \wedge\ \mathcal{X}\mathit{Tmp} = \mathit{normal}) \vee \\ (V'^i_b\ \wedge\ \mathcal{X}\mathit{Tmp} = \mathit{break}) \vee \\ (V'^i_e\ \wedge\ \mathcal{X}\mathit{Tmp} = \mathit{exc}) \end{array}\right),\ V^i_b,\ V^i_e \end{array} \right\}$$

Let B_{Fi} be a *BytecodeProof* for T_{Fi} such that

$$[B_{F_i}, et_{i+1}] = \nabla_S \left(\begin{array}{l} T_{F_i},\ l_{start+i}, l_{start+i+1}, l_{br}, f_{i+1} \ldots f_k, \\ \mathit{divide}(et_i, et'_i[0], l_{start+i}, l_{start+i+1}) \end{array} \right)$$

$$b_{\mathsf{goto}} = \{B^k_b\}\quad l_{start+k+1} : \mathsf{goto}\ l_{br}$$

The definition of the translation is the following:

$$\nabla_S \left(\frac{}{\{P\}\ \ \mathtt{break}\ \ \{\mathit{false}, P, \mathit{false}\}}, l_{start}, l_{next}, l_{br}, f, et_0 \right) \\ = [\ B_{F_1} + B_{F_2} + \ldots B_{F_k} + b_{\mathsf{goto}}, et_k]$$

To argue that the bytecode proof is valid, we have to show that the postcondition of B_{F_i} implies the precondition of $B_{F_{i+1}}$ and that the translation of every block is valid. This is the case because the source rule requires the break-postcondition of s_1 to imply the normal precondition of s_2.

The exception table has two important properties that hold during the translation. The first one (Lemma 1) states that the exception entries, whose starting labels appear after the last label generated by the translation, are kept unchanged. The second one (Lemma 2) expresses that the exception entry is not changed by the division. These properties are used to prove soundness of the translation.

Lemma 1. *If* $\nabla_S(\{P_n\}\ s\ \{Q\},\ l_a,\ l_{b+1},\ l_{break},\ f,\ et) = [(I_{l_a} \ldots I_{l_b}), et']$ *and* $l_{start} \leq l_a < l_b \leq l_{end}$ *then for every* $l_s, l_e \in$ *Label such that* $l_b < l_s < l_e \leq l_{end}$ *and for every* $T \in$ *Type such that* $T \preceq \mathit{Throwable} \vee T \equiv \mathit{any}$, *the following holds:* $et[l_{start}, l_{end}, T] = et'[l_s, l_e, T]$.

Lemma 2. *Let* $r \in \mathit{ExcTableEntry}$ *and* $et' \in \mathit{ExcTable}$ *be such that* $r \in et'$. *If* $et \in \mathit{ExcTable}$ *and* $l_s, l_e \in \mathit{Label}$ *are such that* $et = \mathit{divide}(et', r, l_s, l_e)$, *then* $et[l_s, l_e, T] = r[2]$

5. EXAMPLE

Figure 4 exemplifies the translation. The source proof of the example in Figure 3 is presented on the left-hand side and the corresponding bytecode proof on the right. An exception is thrown in the `try` block with precondition $b = 1$. The `finally` block increases b and then executes a `break` changing the status of the program to break mode (the postcondition is $b = 2$). In the bytecode proof, the body of the loop is between lines 09 and 18. Lines 17 and 18 re-throw the exception produced at line 10. Due to the execution of a `break` instruction, the code from 17 to 18 is not reachable (this is the reason for their *false* precondition). The `break` translation yields at line 16 a goto instruction whose target is the end of the loop, *i.e.*, line 23.

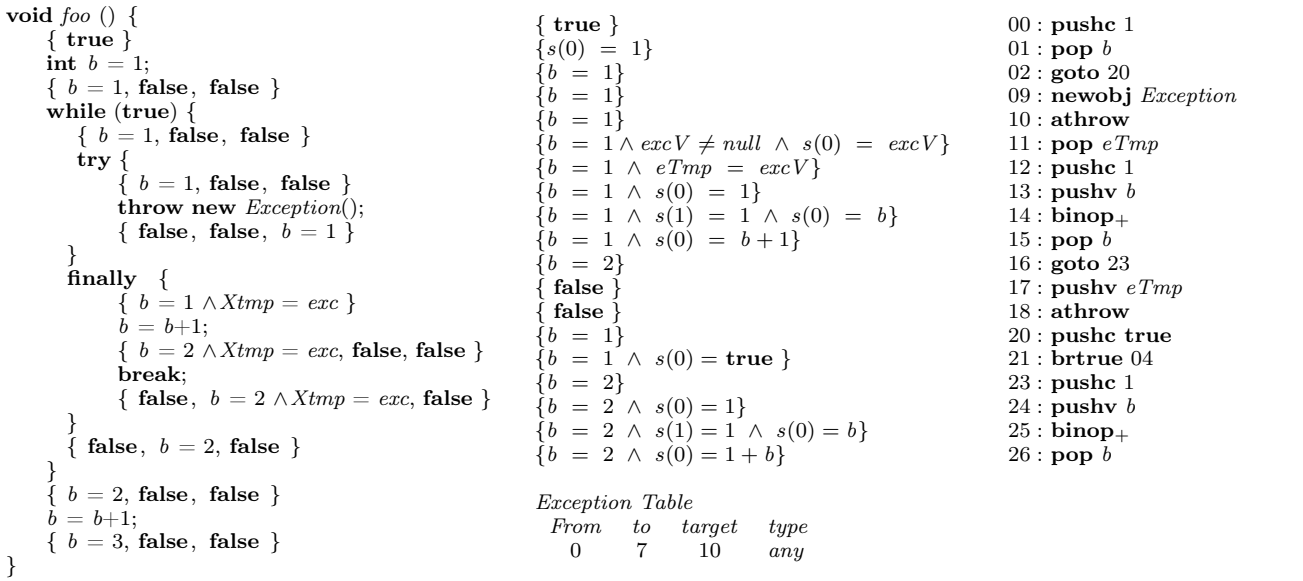


Figure 4: Example of source and bytecode proofs generated by the PTC.

6. SOUNDNESS THEOREM

In a PCC environment, a soundness proof is required only for the trusted components. PTCs are not part of the trusted code base: If the PTC generates an invalid proof, the proof checker would reject it. But from the point of view of the code producer, we would like to have a compiler that always generates valid proofs. Otherwise, it would be useless.

We prove the soundness of the translations, *i.e.*, the translation produces valid bytecode proofs. It is, however, not enough to prove that the translation produces a valid proof, because the compiler could generate bytecode proofs where every precondition is false. The theorem states that if (1) we have a valid source proof for the statement s_1, and (2) we have a proof translation from the source proof that produces the instructions $I_{l_{start}} ... I_{l_{end}}$, their respective preconditions $E_{l_{start}} ... E_{l_{end}}$, and the exception table et, and (3) the exceptional postcondition in the source logic implies the precondition at the target label stored in the exception table for all types T such that $T \preceq Throwable \vee T \equiv any$ but considering the value stored in the stack of the bytecode, and (4) the normal postcondition in the source logic implies the next precondition of the last generated instruction (if the last generated instruction is the last instruction of the method, we use the normal postcondition in the source logic), (5) the break postcondition implies *finallyProperties*. Basically, the *finallyProperties* express that for every triple stored in f, the triple holds and the break postcondition of the triple implies the break precondition of the next triple. And the exceptional postcondition implies the precondition at the target label stored in the exception table et_i but considering the value stored in the stack of the bytecode. Then, we have to prove that every bytecode specification holds ($\vdash \{E_l\}\ I_l$).

In the soundness theorem, we use the following abbreviation: for an exception table et, two labels l_a, l_b, and a type T, $et[l_a, l_b, T]$ returns the target label of the first et's exception entry whose starting and ending labels are less or equal and greater or equal than l_a and l_b, respectively, and whose type is a supertype of T.

Due to space limitations, we present the theorem without the details of the properties satisfied by the finally function f. The proof runs by induction on the structure of the derivation tree for $\{P\}\ s_1\ \{Q_n, Q_b, Q_e\}$. The proof and the complete theorem can be found in our technical report [7].

THEOREM 1.

$$\left(\begin{array}{l} \vdash \dfrac{Tree}{\{P\}\ \ s_1\ \ \{Q_n, Q_b, Q_e\}} \equiv\ T_{S_1}\ \ \wedge \\ \left[(I_{l_{start}} ... I_{l_{end}}), et\right] = \nabla_S\left(T_{S_1},\ l_{start}, l_{end+1}, l_{break}, f, et'\right) \wedge \\ (\forall\ T : Type : (T \preceq Throwable \vee T \equiv any) : \\ \ (Q_e \wedge excV \neq null \wedge\ s(0) = excV) \ \ \Rightarrow\ \ E_{et'[l_{start}, l_{end}, T]}) \wedge \\ \left(Q_n \ \ \Rightarrow\ \ E_{l_{end+1}}\right)\ \ \wedge \\ (Q_b\ \Rightarrow finallyProperties) \end{array}\right)$$

$$\Rightarrow$$

$$\forall\ l\ \in\ l_{start}\ ...\ l_{end} : \vdash \{E_l\}\ I_l$$

7. RELATED WORK

Necula and Lee [9] have developed certifying compilers, which produce proofs for basic safety properties such as type safety. Since our approach supports interactive verification of source programs, we can handle more complex properties such as functional correctness.

The open verifier framework for foundational verifiers [4] verifies untrusted code using customized verifiers. The approach is based on foundation proof carrying code. The architecture consists of a trusted checker, a fixpoint module, and an untrusted extension (a new verifier developed by untrusted users). However, the properties that can be proved are still limited.

A certified compiler [5, 11] is a compiler that generates a proof that the translation from the source program to the assembly code preserves the semantics of the source program. Together with a source proof, this gives an indirect

correctness proof for the bytecode program. Our approach generates the bytecode proof directly, which leads to smaller certificates.

Barthe *et al.* [3] show that proof obligations are preserved by compilation (for a non-optimizer compiler). They prove the equivalence between the verification condition (VC) generated over the source code and the bytecode. The source language is an imperative language which includes method invocation, loops, conditional statements, `try-catch` and `throw` statements. However, they do not consider `try-finally` statements, which make the translation significantly more complex. Our translation supports `try-finally` and `break` statements.

Pavlova [10] extends the aforementioned work to a subset of Java (which includes `try-catch`, `try-finally`, and `return` statements). She proves equivalence between the VC generated from the source program and the VC generated from the bytecode program. The translation of the above source language has a similar complexity to the translation presented in this paper. However, Pavlova avoided the code duplication for `finally` blocks by disallowing `return` statements inside the `try` blocks of `try-finally` statements. This simplifies not only the verification condition generator, but also the translation and the soundness proof.

Furthermore, Barthe *et al.* [2] translate certificates for optimizing compilers from a simple interactive language to an intermediate RTL language (Register Transfer Language). The translation is done in two steps: first the source program is translated into RTL and then optimizations are performed building the appropriate certificate. Barthe *et al.* use a source language that is simpler than ours. We will investigate optimizing compilers as part of future work.

This work is based on Müller and Bannwart's work [1]. They present a proof-transforming compiler from a subset of Java which includes loops, conditional statements and object oriented features. We have extended the source language including exception handling and `break` statements. Moreover, we have also proved soundness.

8. CONCLUSION

We have defined proof transformation from a subset of Java to bytecode. The PTC allows us to develop the proof in the source language (which is simpler), and transforms it into a bytecode proof. Since Java source and bytecode are very similar, proof transformation is simple for many language features. In this paper, we focused on one of the most complex translations, namely the interaction between `try-finally` and `break` statements. We showed that our translation is sound, that is, it produces valid bytecode proofs.

To show the feasibility of our approach, we implemented a PTC for a language similar to the Java subset considered here. The compiler takes a proof in XML format and produces the bytecode proof.

As future work, we plan to extend the source language with statements like `return` and `continue`. Also, we plan to develop a proof checker that tests the bytecode proof. Moreover, we plan to analyze how proofs can be translated using an optimizing compiler.

Moreover, we will investigate proof-transforming compilation for language features that cannot by directly mapped to bytecode such as multiple inheritance and Eiffel's once methods. This extension will lead to a more general transformation framework.

9. ACKNOWLEDGMENTS

We would like to thank Nicu Georgian Fruja for reviewing and providing helpful comments on drafts of this paper. Müller's work was carried out at ETH Zurich. It was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-2005-015905 MOBIUS project. Nordio's work was funded in part by ETH under the Heterogeneous Proof-Carrying Components project.

10. REFERENCES

[1] F. Y. Bannwart and P. Müller. A Logic for Bytecode. In F. Spoto, editor, *Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE)*, volume 141 of *ENTCS*, pages 255–273. Elsevier, 2005.

[2] G. Barthe, B. Grégoire, C. Kunz, and T. Rezk. Certificate Translation for Optimizing Compilers. In *13th International Static Analysis Symposium (SAS)*, LNCS, Seoul, Korea, August 2006. Springer-Verlag.

[3] G. Barthe, T. Rezk, and A. Saabas. Proof obligations preserving compilation. In *Third International Workshop on Formal Aspects in Security and Trust, Newcastle, UK*, pages 112–126, 2005.

[4] B. Chang, A. Chlipala, G. Necula, and R. Schneck. The Open Verifier Framework for Foundational Verifiers. In *ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI'05)*, 2005.

[5] G. Goos and W. Zimmermann. Verification of Compilers. LNCS, pages 201–230. Springer-Verlag, 2005.

[6] P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *LNCS*. Springer-Verlag, 2002.

[7] P. Müller and M. Nordio. Proof-Transforming Compilation of Programs with Abrupt Termination. Technical Report 565, ETH Zurich, 2007.

[8] G. Necula. *Compiling with Proofs*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1998.

[9] G. Necula and P. Lee. The Design and Implementation of a Certifying Compiler. In *Programming Language Design and Implementation (PLDI)*, pages 333–344. ACM Press, 1998.

[10] M. Pavlova. *Java Bytecode verification and its applications*. PhD thesis, University of Nice Sophia-Antipolis, 2007.

[11] A. Poetzsch-Heffter and M. J. Gawkowski. Towards Proof Generating Compilers. *ENTCS*, 132(1):37–51, 2005.

[12] A. Poetzsch-Heffter and P. Müller. A Programming Logic for Sequential Java. In S. D. Swierstra, editor, *European Symposium on Programming Languages and Systems (ESOP'99)*, volume 1576 of *LNCS*, pages 162–176. Springer-Verlag, 1999.

[13] A. Poetzsch-Heffter and N. Rauch. Soundness and Relative Completeness of a Programming Logic for a Sequential Java Subset. Technical report, Technische Universität Kaiserlautern, 2004.

An Integrated Verification Environment for JML: Architecture and Early Results

Patrice Chalin, Perry R. James, George Karabotsos
Dependable Software Research Group,
Dept. of Computer Science and Software Engineering,
Concordia University, Montréal, Canada
{chalin, perry, g_karab}@dsrg.org

ABSTRACT

Tool support for the Java Modeling Language (JML) is a very pressing problem. A main issue with current tools is their architecture: the cost of keeping up with the evolution of Java is prohibitively high: e.g., almost three years following its release, Java 5 has yet to be fully supported. This paper presents the architecture of JML4, an Integrated Verification Environment (IVE) for JML that builds upon Eclipse's support for Java, enhancing it with Extended Static Checking (ESC), an early form of Runtime Assertion Checking (RAC) and JML's non-null type system. Early results indicate that the synergy of complementary verification techniques (being made available within a single tool) can help developers be more effective; we demonstrate new bugs uncovered in JML annotated Java source—like ESC/Java2—which is routinely verified using first generation JML tools.

Categories and Subject Descriptors

D.2.4 [Software Engineering]: Software/Program Verification—programming by contract; D.3.3 [Programming Languages]; F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs.

General Terms

Design, Languages, Theory, Verification.

Keywords

Integrated Verification Environment, Java Modeling Language, Eclipse, JML4.

1. INTRODUCTION

The Java Modeling Language (JML) is the most popular Behavioral Interface Specification Language (BISL) for Java. JML is recognized by a dozen tools and used by over two dozen institutions for teaching and/or research, mainly in the context of program verification [18]. Tools exist to support the full range of verification from runtime assertion checking (RAC) to full static program verification (FSPV) with extended static checking (ESC) in between [3]. Of these, RAC and ESC are the technologies which are most likely to be adopted by mainstream developers because of their ease of use and low learning curve.

Sixth International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2007), September 3-4, 2007, Cavtat near Dubrovnik, Croatia.

In earlier work [6] we confirmed (among other things) how RAC and ESC are most effective when used *together*, particularly when it comes to the verification of sizeable systems. Unfortunately, this is more challenging than it should be; one of the key reasons being that the tools accept slightly different and incompatible variants of JML—sadly this is the case for practically all of the current JML tools. The top factors contributing to the current state of affairs are

- partly historical—the tools were developed independently, each having their own parsers, type checkers, etc. and
- partly due to the rapid pace of evolution of both JML and Java.

Not only does this last point make it difficult for individual research teams to keep apace, it also results in significant and unnecessary duplication of effort.

For some time now the JML community has recognized that a consolidation effort is necessary with respect to its tool base. In response to this need, three prototypical "next generation" tools have taken shape: JML3, JML4, and JML5 [18]. This paper presents the architecture and design rationale behind JML4: we explain why we believe JML4 will not suffer from the maintenance overhead of other JML tools even in the face of the rapid pace of evolution of Java.

The remainder of the paper is organized as follows. In the next section, we present early results demonstrating that the synergy of complementary verification techniques (being made available within JML4) can help developers be more effective; we illustrate new bugs uncovered in ESC/Java2 source—despite the fact that the code is routinely verified using itself and other JML tools. The remaining sections focus on JML tool support, offering

- a discussion of the goals to be achieved by any next generation JML tool base (Section 3) and
- a presentation (Section 4) of the architectural and (some aspects of) the detailed design of JML4; our objective is to provide sufficient detail to allow JML4's design to be assessed relative to the stated goals.

Section 5 provides initial arguments supporting our belief that JML4's design will be less costly to maintain in the long run than current JML tools. Section 6 offers a brief discussion and comparison of JML4 with its predecessor JML2 and siblings JML3 and JML5 as well as other tools like the Java Applet Correctness Kit (JACK). Conclusions and future work are presented in Section 7.

2. EARLY RESULTS: BENEFITS OF SYNERGY

One of JML4's first and most fully developed features is JML's non-null type system [7]. This, coupled with the tool's ability to read the extensive JML API library specifications, renders it quite effective at statically detecting potential null pointer exceptions (NPEs). Recently, JML4 was enhanced to

```
package escjava;
...
public class Main extends javafe.SrcTool {
 ...
 public static Options options() {
   return (Options)options;
 }
 ...
 public String processRoutineDecl(...) {
   ...
   VcGenerator vcg = null; ...
   try {
      ... // possible assignment to vcg
   } // multiple catch blocks
   catch (Exception e) {
      ...
   }
   ...
   fw.write(vcg.old2Dot()); // <<< possible NPE
   ...
 }
}
```

Figure 1. Code excerpt from the escjava.Main class

support Extended Static Checking (ESC) through the integration of ESC/Java2 [11]. While each verification technique has strengths and weaknesses, integration of complementary techniques into a single verification environment brings about a level of synergy that would not be achievable otherwise.

As a concrete example of the kind of verification technique synergy which JML4 achieves, consider the code fragment given in Figure 1, an excerpt from ESC/Java2's `escjava.Main` class. JML4 correctly reports that a dereference of `vcg` inside of `processRoutineDecl()` could result in an NPE (Figure 2).

Since ESC/Java2 is routinely run on itself, why was this error not detected before? Because analyzing `processRoutine-Decl()`, which consists of 386 lines of code, is beyond the capabilities of ESC/Java2 (it gives up on attempting to verify the method because the verification condition is too big). Several errors that arise under such circumstances were identified in ESC/Java2 source by JML4.

As another example, consider the static `options()` method of `escjava.Main` (Figure 1) which returns a reference to ESC/Java2's command line options. This method is used throughout the code (272 occurrences) and its return value is directly dereferenced even though the method can return null.

While JML4 reports the 250+ NPEs related to the use of this method, ESC fails to do so because another ESC error prevents it from determining that the method can return null: namely, a possible type cast violation. The effect of having one error mask others is particularly acute for ESC/Java2 (even more so than in ordinary compilers) thus making effective the more resilient, though less powerful, complementary verification capabilities of other techniques such as those implemented in JML4 (and recently added to ESC/Java2 [17]). Our preliminary use of JML4 has demonstrated that, e.g., nullity type errors once fixed allow ESC to push further its analysis, helping expose yet more bugs in code and specifications, which leads to uncovering further nullity type errors, etc.

3. JML Tools: Background and Goals

In this section we discuss the main goals to be satisfied by any next generation tool base for JML. Before doing so we give a brief summary of the JML's first generation of tools.

3.1 First Generation Tools

The first generation JML tools essentially consist of:

- Common JML tool suite—formerly the Iowa State University (ISU) JML tool suite—also known to developers as JML2, which includes the JML RAC compiler and JmlUnit [3],
- ESC/Java2, an extended static checker [11], and
- LOOP a full static program verifier [20].

Of these, JML2 is the original JML tool set. Although ESC/Java2 and LOOP initially used an annotation language other than JML, they quickly switched to use JML.

Being independent development efforts, each of the tools mentioned above has its own Java/JML front end including scanner, parser, abstract syntax tree (AST) hierarchy and static

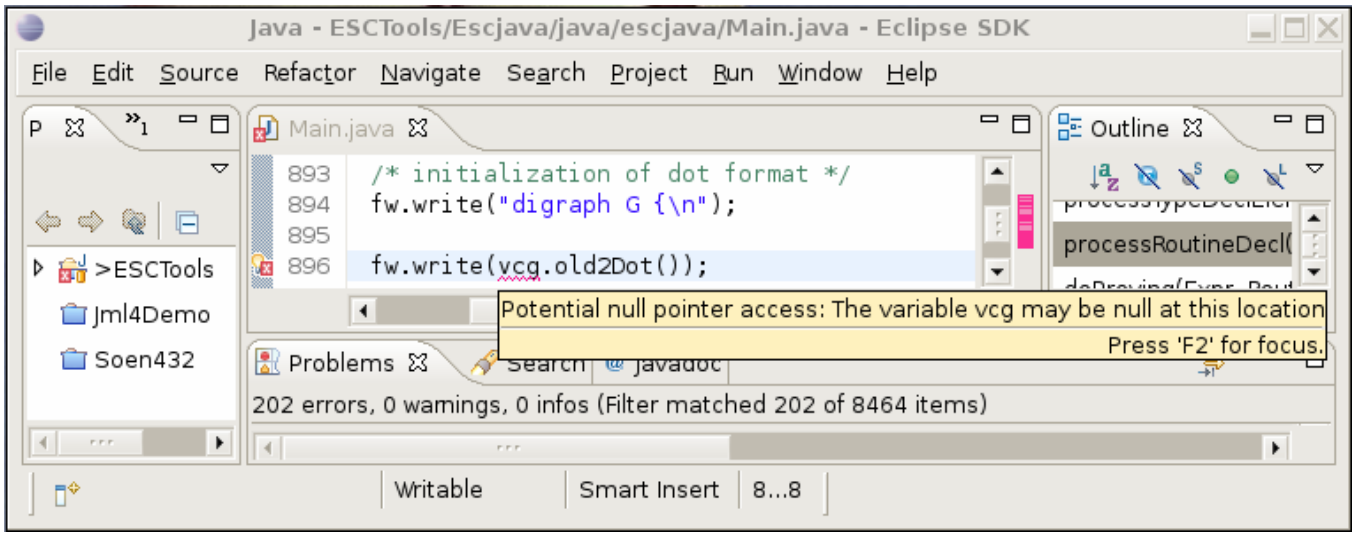


Figure 2. JML4 reporting non-null type system errors in a method too big for ESC to verify

analysis code—though not all developed to the same level of completeness or reliability. This is a considerable amount of duplicate effort and code (of the order of 50-100K SLOC[1]). This became evident as JML evolved, but the main hurdle which has yet to be fully addressed is the advent of Java 5 (especially generics).

3.1.1 Lessons Learned from JML2

Which lessons can be learned from the development of the first generation of tools, especially JML2 which, from the start, has been the reference implementation of JML? JML2 was essentially developed as an extension to the MultiJava (MJ) compiler. By "extension", we mean that

- for the most part, MJ remains independent of JML
- many JML features are naturally implemented by subclassing MJ features and overriding methods—e.g. abstract syntax tree nodes with their associated type checking methods;
- in other situations, extension points (calls to methods with empty bodies) were added to MJ classes so that it was possible to override behavior in JML2.

We believe that this approach has allowed JML2 to be successfully maintained as the JML reference implementation since 2002 by an increasing developer pool (there are currently 49 registered developers). In that case what, if anything, went wrong? We believe it was a combination of factors including the advent of a relatively big step in the evolution of Java (including Java 5 generics) and the difficulty in finding developers to upgrade MJ. Hence our approach in JML4 has been to repeat the successful approach adopted by JML2 but to ensure that we choose to extend a Java compiler that we are confident will be maintained (outside of the JML community).

3.1.2 Evolution of IDEs

Another important point to be made about the first generation of JML tools is that they are mainly command line tools, though some developers were able to make comfortable use of them inside Emacs, which in a sense, can be considered an early integrated development environment (IDE).

With a phenomenal increase in the popularity of modern IDEs like Eclipse, it seems clear that to increase the likelihood of getting widespread adoption of JML, it will be necessary to have its tools operate well within one or more popular IDEs. In recognition of this, early efforts have successfully provided basic JML tool support via Eclipse plug-ins, which mainly offer access to the command line capabilities of the JML RAC or ESC/Java2.

Other efforts (generation 1.5), resulted in tools that were built from the outset within an IDE but have *not* been designed to support RAC and ESC. These include the

- Java Applet Correctness Kit (JACK), built directly as an Eclipse plug-in, supports interactive static verification [2].
- KeY tool, which was recently adapted to support JML as a constraint language for expressing specifications in design models. The KeY tool is built on top of Borland's Together IDE [1, 12].

[1] (Physical) Source Lines of Code obtained by counting end-of-lines for non-comment code.

3.2 Goals For Next Generation Tool Bases

We are targeting mainstream industrial software developers as our key end users. From an end user point of view, we strive to offer a single Integrated (Development and) Verification Environment (IVE) within which they can use any desired combination of RAC, ESC, and FSPV technology. No single tool currently offers this feature set for JML. In addition, user assistance by means of the auto-generation of specifications (or specification fragments) should be possible—e.g. based on approaches currently offered by tools like Daikon [14], Houdini [15] and JmlSpec [3].

Since JML is essentially a superset of Java, most JML tools will require, at a minimum, the capabilities of a Java compiler front end. Some tools (e.g., the RAC) would benefit from compiler back-end support as well. One of the important challenges faced by the JML community is keeping up with the rapid pace of the evolution of Java. As researchers in the field of applied formal methods, we get little or no reward for developing and/or maintaining basic support for Java. While such support is essential, it is also very labor intensive. Hence, an ideal solution would be to extend a Java compiler, already integrated within a modern IDE, whose maintenance is assured by a developer base outside of the JML research community. If the extension points can be judiciously chosen and kept to a minimum then the extra effort caused by developing on top of a rapidly moving base can be minimized.

In summary, our general goals are to provide

- a base framework for the integrated capabilities of RAC, ESC, and FSPV
- in the context of a modern Java IDE whose maintenance is outside the JML community
- by implementing support for JML as extensions to the base support for Java so as to minimize the integration effort required when new versions of the IDE are released.

A few recent projects have attempted to satisfy these goals. In the next section, we describe how we have attempted to satisfy them in our design of JML4; the other projects are discussed in the section on related work.

4. JML4

In our first feature set, JML4 enhanced Eclipse 3.3 with: scanning and parsing of nullity modifiers, enforcement of JML's non-null type system (both statically and at runtime) and the ability to read and make use of the extensive JML API library specifications. This subset of features was chosen so as to exercise some of the basic capabilities that any JML extension to Eclipse would need to support. These include

- recognizing and processing JML syntax inside specially marked comments, both in `*.java` files as well as `*.jml` files;
- storing JML-specific nodes in an extended AST hierarchy,
- statically enforcing a modified type system, and
- providing for runtime assertion checking (RAC).

Also, the chosen subset of features is useful in its own right, somewhat independent of other JML features [7]; i.e. the capabilities form a natural extension to the existing embryonic Eclipse support for nullity analysis.

We have since been pursuing our enrichment of the JML4 feature set so that to date, we have completed a full integration of

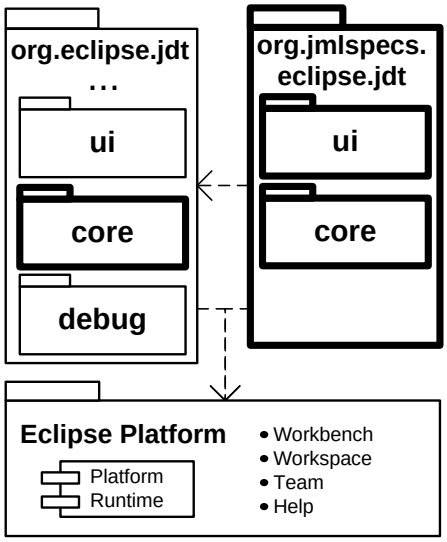


Figure 3. High-level package view

ESC/Java2 and begun work towards the support of runtime assertion checking of JML Level 0 [19, Section 2.9].

In the remainder of this section, we present our proposed means of extending Eclipse to support JML, appealing at times to the specific way in which the JML4 features described above have been realized.

4.1 Architectural Overview

Eclipse is a plug-in based application platform. An Eclipse application consists of the Eclipse plug-in loader (Platform Runtime component), certain common plug-ins (such as those in the Eclipse Platform package) along with application specific plug-ins. Well known bundlings of Eclipse plug-ins include the Eclipse Software Development Kit (SDK) and the Eclipse Rich Client Platform (RCP). While Eclipse is written in Java, it does not have built-in support for Java. Like all other Eclipse features, Java support is provided by a collection of plug-ins—called the Eclipse Java Development Tooling (JDT)—offering, among other things, a standard Java compiler and debugger.

The main packages of interest in the JDT are the `ui`, `core`, and `debug`. As can be gathered from the names, the core (non-UI) compiler functionality is defined in the `core` package; UI elements and debugger infrastructure are provided by the components in the `ui` and `debug` packages, respectively.

One of the rules of Eclipse development is that public APIs must be maintained *forever*. This API stability helps avoid breaking client code. The following convention was established by Eclipse developers: only classes or interfaces that are *not* in a package named `internal` can be considered part of the *public API*. Hence, for example, the classes for the JDT's internal AST are found in the `org.eclipse.jdt.internal.compiler.ast` package, where as the public version of the AST is (partly) reproduced under `org.eclipse.jdt.core.dom`. For JML4 we have generally made changes to internal components (to insert hooks) and then moved most of the JML specific code to `org.jmlspecs.eclipse.jdt`.

At the top-most level, JML4 consists of:

- a customized version of the `org.eclipse.jdt.core` package (details will be given below) that is used as a drop-in replacement for the official Eclipse JDT `core`.

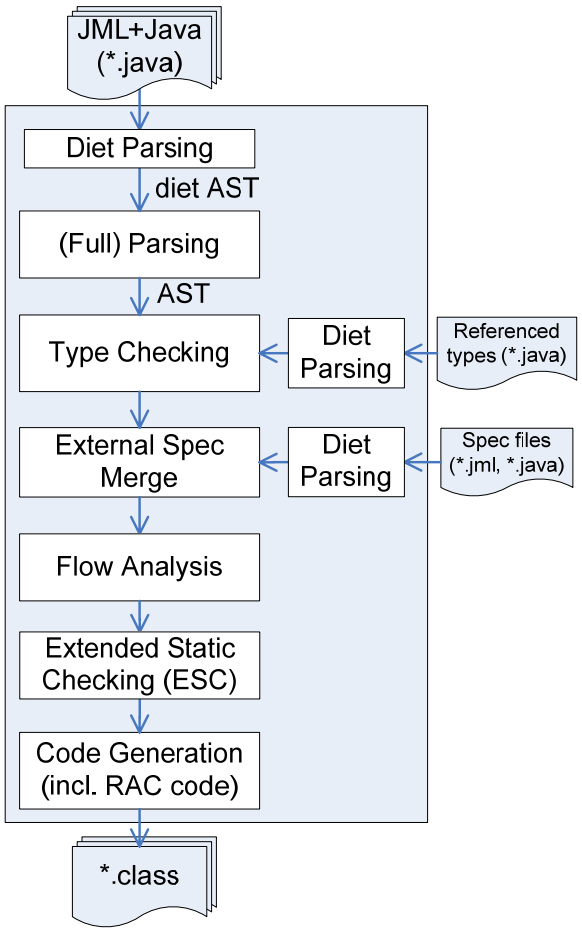


Figure 4. JDT/JML4 compilation phases

- JML specific classes contained in `org.jmlspecs.eclipse.jdt` including core classes (most of which are subclasses of the JDT Abstract Syntax Tree (AST) node hierarchy) and ui classes (e.g. for JML related preferences).

These packages are shown in bold in Figure 3.

4.2 Compilation Phases Overview

The main steps of the compilation process performed by JML4 are illustrated in Figure 4. In the Eclipse JDT (and JML4), there are two types of parsing: in addition to a standard full parse, there is also a diet parse, which only gathers signature information and ignores method bodies. When a set of JML annotated Java files is to be compiled, all are diet parsed to create (diet) ASTs containing initial type information, and the resulting type bindings are stored in the lookup environment (not shown). Then each compilation unit (CU) is fully parsed. During the processing of each CU, types that are referenced but not yet in the lookup environment must have type bindings created for them. This is done by first searching for a binary (`*.class`) file or, if not found, a source (`*.java`) file. Bindings are created directly from a binary file, but a source file must be diet parsed and added to the list to be processed. In both cases the bindings are added to the lookup environment. If JML specifications for any CU or referenced type are contained in a separate external file (e.g. a `*.jml` file), then these specification files are diet parsed and the resulting information merged with the CU AST (or associated with the

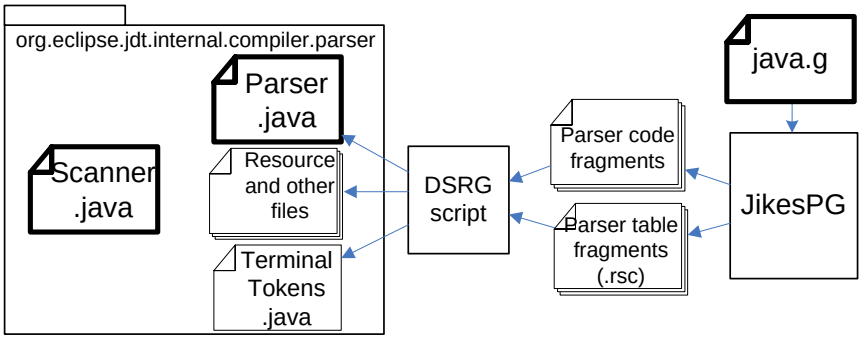


Figure 5. Customizing the JDT lexer and parser

binding in the case of a binary file). Finally, flow analysis and code generation are performed. Extended static checking is treated as a distinct phase between flow analysis and code generation. In the remaining subsections we briefly cover some aspects of JML4 compilation—details can be found in [8].

4.3 Lexical Scanning, Parsing and the AST

Figure 5 provides an overview of the main parser components as well as the means by which they are generated; components in bold are those that have been customized under JML4.

Scanning. Since all of JML is contained within specially marked comments, the main change to the lexical scanner was to enhance it to recognize JML annotations. This is currently handled using a Boolean field that indicates if the scanner is in a JML annotation or not. Adding support for new keywords requires a little more work than usual since the JDT's scanner is highly optimized and hand crafted. Keywords, for example, are identified by a set of nested case statements based on the first character of a lexeme and its length.

Parsing. The JDT's parser is auto-generated from a grammar file (`java.g`) using the Jikes Parser Generator (JikesPG) and a custom script we have written. On a positive note, the grammar file, `java.g`, closely follows the Java Language Specification [16] and hence has been relatively easy to extend. Possibly the main source of difficulty in the parser is the lack of automatic support for token stacks.

Other than adding methods corresponding to new grammar-rule reductions, the most prominent change to the parser is the replacement of calls to constructors of JDT AST nodes with those of JML-specific AST subclasses. The abstract syntax tree hierarchy for JML4 is obtained by subclassing specific JDT AST nodes as needed. An illustration of how this is done is given in Figure 6. For example, JML type references are like Java type references but have additional information such as nullity. Currently we subclass 20% of the AST node types; the JML specific subclasses generally contain very little code (and in particular, no code is copied from superclasses).

4.4 Type Checking and Flow Analysis

Type checking is performed by invoking the `resolve()` method on a compilation unit. Similarly, flow analysis is performed by the `analyseCode()` method. Addition of JML functionality is achieved by inserting "hooks" into the previously mentioned methods—i.e. calls to methods with empty bodies in the parent class that are then overridden in JML-specific AST nodes. Our hope is that such hooks will be ported back into the Eclipse JDT, something the JDT developers have confirmed is feasible provided we can demonstrate that no public APIs are changed and that there is little or no impact on runtime performance.

Between type checking and flow analysis, the compiler checks for external specification files (e.g., `*.jml` files) corresponding to the file being compiled. If one is found, it is parsed and any annotations are added to the corresponding declarations. Binary types (i.e., those found in `*.class` files) whose specifications are needed are handled differently. For these, the system searches for both a source and external specification file.

4.5 Runtime Checking and Extended Static Checking

Code generation is performed by each `ASTNode`'s `generateCode()` method. Its `CodeStream` parameter provides methods for emitting JVM bytecode and hides some of the bookkeeping details, such as determining the generated code's runtime stack usage. Hence, supporting runtime checking is relatively straightforward.

Extended Static Checking in JML4 is currently achieved by a

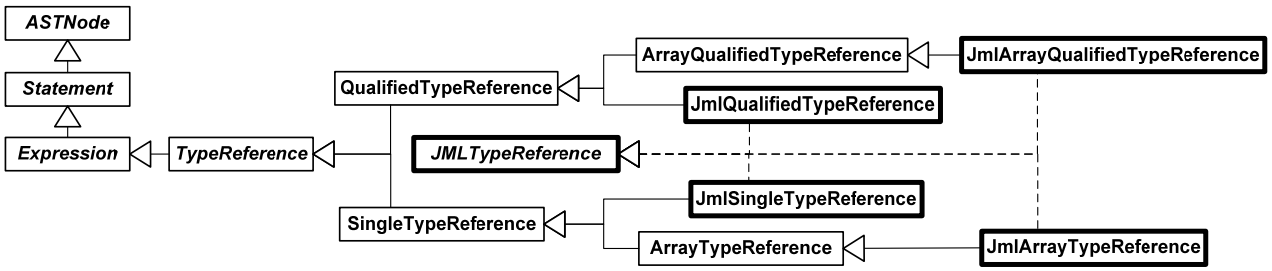


Figure 6. Part of the AST hierarchy (`org.eclipse.jdt.internal.compiler.ast`)

preliminary integration of ESC/Java2. That is, during compilation in a step just following flow analysis, we invoke escjava's main processing method—which effectively reparses the file inside of ESC/Java2. While such an approach is inefficient, it has allowed us to focus on the integration of the problem reporting. As a next step, we will create a visitor which will map Eclipse JDT AST's into ESC/Java2's AST, thus avoiding the reparsing. Finally, we plan on building a custom transformation from the JDT's AST into ESC/Java2's guarded command language, hopefully allowing us to reuse the rest of ESC/Java2's verification condition generation back-end.

5. Validation of Architectural Approach

JML4 was recently used to help validate our proposal that JML's non-null type system should be non-null by default [7]. It was used to produce RAC-enabled versions of five case studies (totaling over 470K SLOC), which were then used to execute those systems' extensive test suites. This exercise gave us confidence in JML4's runtime checking capabilities and its ability to process JML API specifications.

JML4, like JML2, is built as a closely integrated and yet loosely coupled extension to an existing compiler. An additional benefit for JML4 is that the timely compiler base maintenance is assured by the Eclipse Foundation developers. Hence, as compared to JML2, we have traded in committer rights for free maintenance; a choice which we believe will be more advantageous in the long run—in particular due to the rapid pace of the evolution of Java. Unfortunately, loosing committer rights means that we must maintain our own version of the JDT code. Use of the CVS vendor branch feature has made this manageable.

While we originally had the goal of creating JML4 as a proper Eclipse plug-in, only making use of public JDT APIs (rather than as a replacement plug-in for the JDT), it rapidly became clear that this would result in far too much copy-and-change code; so much so that the advantage of coupling to an existing compiler was lost (e.g. due to the need to maintain our own full parser and AST).

Nonetheless we were also originally reluctant to build atop internal APIs, which contrary to public APIs, are subject to change—with weekly releases of the JDT code, it seemed like we would be building on quicksand. Anticipating this, we established several conventions that make merging in the frequent JDT changes both easier and less error prone. These include

- avoiding introducing JML features by the copy-and-change of JDT code, instead we make use of subclassing and method extension points;
- bracketing any changes to our copy of the JDT code with special comment markers.

While following these conventions, incorporating each of the regular JDT updates since the fall of 2006 (to our surprise) has taken less than 10 minutes, on average.

6. Related Work

In this section we briefly compare JML4 to its sibling next generation projects JML3, JML5 as well as to the Java Applet Correctness Kit (JACK). Further details, examples and tools are covered in [8].

The first next-generation Eclipse-based initiative was JML3, created by David Cok. The main objective of the project was to create a proper Eclipse plug-in, independent of the internals of the JDT [9]. Considerable work has been done to develop the necessary infrastructure, but there are growing concerns about the long term costs of this approach.

Because the JDT's parser is not extensible from public JDT extensions points, a separate parser for the entire Java language and an AST had to be created for JML3; in addition, Cok notes that "JML3 [will need] to have its own name / type / resolver / checker for both JML constructs [and] all of Java" [9]. Since one of the main goals of the next generation tools is to escape from providing support for the Java language, this is a key disadvantage.

The Java Applet Correctness Kit (JACK) is a proprietary tool for JML annotated Java Card programs initially developed at Gemplus (2002) and then taken over by INRIA (2003) [2]. It uses a weakest precondition calculus to generate proof obligations that are discharged automatically or interactively using various theorem provers [5]. While JACK is emerging as a candidate next generation tool (offering features unique to JML tools such as verification of annotated byte code [4] and a proof obligation viewer), being a proper Eclipse plug-in, it suffers from the same drawbacks as JML3. Additionally JACK does not provide support for RAC which we believe is an essential component of a mainstream IVE.

The JML5 project, recently initiated at Iowa State University, has taken a different approach. Its goal is to embed JML specifications in Java 5 annotations rather than Java comments. Such a change will allow JML's tools to use any Java 5 compliant compiler. Unfortunately, the use of annotations has important drawbacks as well. In addition to requiring a separate parser to process the JML specific annotation contents (e.g. assertion expressions), Java's current annotation facility does not allow for annotations to be placed at all locations in the code at which JML can be placed. JSR-308 is addressing this problem as a consequence of its mandate, but any changes proposed would only be present in Java 7 and would not allow support for earlier versions of Java [13].

Table 1 presents a summary of the comparison of the tools. As compared to the approach taken in JML4, the main drawback of the other tools is that they are likely to require more effort to maintain over the long haul as Java continues to evolve and due to the looser coupling with their base.

7. Conclusion and Future Work

The idea of providing JML tool support by means of a closely integrated and yet loosely coupled extension to an existing compiler was successfully realized in JML2. This has worked well since 2002, but unfortunately the chosen Java compiler is not being kept up to date with respect to Java in a timely manner. We propose applying the same approach by extending the Eclipse JDT (partly through internal packages). Even though it is more invasive than a proper plug-in solution, using this approach we have demonstrated that it was relatively easy to enhance the type system and provide RAC and ESC support.

Table 1. A Comparison of Possible Next Generation JML Tools

		JML2	JML3	JML4	JML5	ESC/Java2 Plug-in	JACK
Base Compiler / IDE	**Name**	MJ	JDT	JDT	any Java 7+	ESC/Java2 and JDT	JDT
	Maintained (supports Java ≥ 5)	×	✓	✓	✓	×[1]	✓
Reuse/extension of base (e.g. parser, AST) vs. copy-and-change		✓	×	✓	×	×	×
Tool Support	**RAC**	✓	✓	✓	(✓)	N/A	N/A
	ESC	N/A	(✓)	✓	N/A	✓	✓
	FSPV	N/A	(✓)	(✓)	N/A	N/A	✓

MJ = MultiJava, JDT = Eclipse Java Develoment Toolkit
N/A = not possible, practical or not a goal, (✓) = planned

[1] ESC/Java2 is currently being maintained to support new verification functionality, but its compiler front end has yet to reach Java 5 [10].

Other possible next generation JML tools have been discussed, but all seem to share the common overhead of maintaining a full Java parser, AST, and type checker separate from the base tools they are built from. This seems like an overhead that will be too costly in the long run. We are certainly not claiming that JML4 is the only viable next generation candidate but are hopeful that this paper has demonstrated that it *is* a likely candidate.

REFERENCES

[1] W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. H. Schmitt, "The KeY Tool", *Software and System Modeling*, 4:32-54, 2005.

[2] G. Barthe, L. Burdy, J. Charles, B. Grégoire, M. Huisman, J.-L. Lanet, M. Pavlova, and A. Requet, "JACK: a tool for validation of security and behaviour of Java applications". *Proceedings of the 5th International Symposium on Formal Methods for Components and Objects (FMCO)*, 2007.

[3] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll, "An Overview of JML Tools and Applications", *International Journal on Software Tools for Technology Transfer (STTT)*, 7(3):212-232, 2005.

[4] L. Burdy, M. Huisman, and M. Pavlova, "Preliminary Design of BML: A Behavioral Interface Specification Language For Java Bytecode". *Proceedings of the Fundamental Approaches to Software Engineering (FASE)*, vol. 4422 of *LNCS*, pp. 215-229, 2007.

[5] L. Burdy, A. Requet, and J.-L. Lanet, "Java Applet Correctness: A Developer-Oriented Approach". *Proceedings of the International Symposium of Formal Methods Europe*, vol. 2805 of *LNCS*. Springer, 2003.

[6] P. Chalin and P. James, "Cross-Verification of JML Tools: An ESC/Java2 Case Study". *Proceedings of the Workshop on Verified Software: Theories, Tools, and Experiments (VSTTE)*, Seattle, Washington, August, 2006.

[7] P. Chalin and P. James, "Non-null References by Default in Java: Alleviating the Nullity Annotation Burden". *Proceedings of the 21st European Conference on Object-Oriented Programming (ECOOP)*, Berlin, Germany, July-August, 2007.

[8] P. Chalin, P. R. James, and G. Karabotsos, "The Architecture of JML4, a Proposed Integrated Verification Environment for JML", Dependable Software Research Group, Concordia University, ENCS-CSE-TR 2007-006. May, 2007.

[9] D. R. Cok, "Design Notes (Eclipse.txt)", http://jmlspecs.svn.sourceforge.net/viewvc/jmlspecs/trunk/docs/eclipse.txt, 2007.

[10] D. R. Cok, E. Hubbers, and E. Rodríguez, "Esc/Java2 Eclipse Plug-in", http://sort.ucd.ie/projects/escjava-eclipse/, 2007.

[11] D. R. Cok and J. R. Kiniry, "ESC/Java2: Uniting ESC/Java and JML". In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean editors, *Proceedings of the International Workshop on the Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS'04)*, Marseille, France, March 10-14, vol. 3362 of *LNCS*, pp. 108-128. Springer, 2004.

[12] C. Engel and A. Roth, "KeY Quicktour for JML": www.key-project.org, 2006.

[13] M. Ernst and D. Coward, "Annotations on Java Types", JCP.org, JSR 308. October 17, 2006.

[14] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The Daikon system for dynamic detection of likely invariants", *Science of Computer Programming*, 2007.

[15] C. Flanagan and K. R. M. Leino, "Houdini, an Annotation Assistant for ESC/Java". *Proceedings of the International Symposium of Formal Methods Europe*, Berlin, Germany, vol. 2021, pp. 500-517. Springer, 2001.

[16] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java Language Specification*, 3rd ed. Addison-Wesley Professional, 2005.

[17] M. Janota, R. Grigore, and M. Moskal, "Reachability Analysis for Annotated Code", UCD Dublin, submitted to SAVCBS, 2007.

[18] G. T. Leavens, "The Java Modeling Language (JML)": http://www.jmlspecs.org, 2007.

[19] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, and P. Chalin, "JML Reference Manual", http://www.jmlspecs.org, 2007.

[20] J. van den Berg and B. Jacobs, "The LOOP compiler for Java and JML". In T. Margaria and W. Yi editors, *Proceedings of the Tools and Algorithms for the Construction and Analysis of Software (TACAS)*, vol. 2031 of *LNCS*, pp. 299-312. Springer, 2001.

Playing with Time in Publish-Subscribe using a Domain-Specific Model Checker

Luciano Baresi, Giorgio Gerosa, Carlo Ghezzi, and Luca Mottola
Dipartimento di Elettronica e Informazione, Politecnico di Milano, Italy
{baresi, gerosa, ghezzi, mottola}@elet.polimi.it

ABSTRACT

Thanks to the sharp decoupling it fosters, the Publish-Subscribe paradigm is particularly suited to the implementation of dynamic applications where components join and leave the system unpredictably, and their distributed interactions change over time. Although this feature represents an asset during the implementation phases, it is usually difficult to reason on the global behavior at design time. The problem is exacerbated by the variety of Publish-Subscribe systems available that greatly differ in the guarantees provided, e.g., in terms of message reliability or delivery order.

Some of the authors already tackled the problem with a domain-specific model checker, whose internals are customized depending on the guarantees assumed on the communication infrastructure. However, we essentially disregarded the timing aspects, which are nonetheless pivotal in many applications exploiting a Publish-Subscribe infrastructure. In this paper we augment our tool to verify temporal properties, and explore the interplay between time and different Publish-Subscribe semantics through a case study. Moreover, we report on an effort to formally verify the correctness of the temporal extension, in an attempt to provide a strong foundation for the results obtained using our tool.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Model checking*; D.2.11 [**Software Engineering**]: Software Architectures—*Patterns*

General Terms

Modeling, verification, distributed architectures.

Keywords

Model checking, Publish-Subscribe, time.

1. INTRODUCTION

In recent years, the rise of pervasive and embedded applications has increasingly demanded for highly dynamic and reconfigurable software architectures. In these scenarios, the application components require the ability to federate spontaneously, and dynamically change the nature of their interactions as new requirements arise. As a result, traditional architectural paradigms (e.g., client-server) are ill-suited to the requirements at hand. In contrast, the Publish-Subscribe (P/S) [13] paradigm provides an asynchronous, implicit, and multi-point communication style that well adapts to dynamic scenarios. As illustrated in Figure 1, in a P/S system components *subscribe* to specific message (event) patterns, and are *notified* when other components *publish* messages matching their subscriptions. A *dispatcher* mediates the communication by storing subscriptions in a dedicated table, and matching them against published messages. Based on this interaction pattern, P/S systems have been developed for a wide range of scenarios, from wide-area notification services [8] to wireless sensor networks [18].

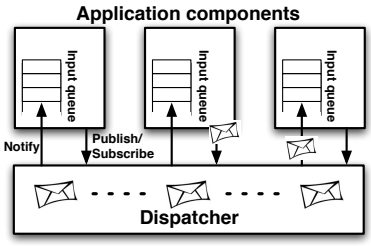


Figure 1: P/S architecture.

Although the P/S paradigm makes it easier to implement dynamic applications, the strong decoupling it fosters renders the global interactions among components difficult to capture and to reason about. This ultimately hinders the verification and validation of the overall federation. Moreover, available P/S systems provide radically different guarantees that may affect the outcome of the verification effort. For instance, different message delivery orderings may have an impact on a component's execution flow, which may reflect in a different system-wide behavior.

To address these issues, model checking has been proposed as a tool to analyze the behavior of applications built on top of P/S infrastructures, e.g., as in [14]. These approaches, however, do not capture many of the guarantees provided by existing P/S systems, thus limiting their applicability. In [2], we proposed a novel approach to the problem: instead of using standard tools, we leverage off the extensible model checker Bogor [19], and augment its input language and internal mechanisms to include P/S operations as primitive constructs. By doing so, we can model the various P/S guarantees *within* the checking engine, and customize the verification based on a specific incarnation of the P/S paradigm. This approach, summarized in Section 2, allows us to achieve a *domain-specific, state abstraction* mechanism, which dramatically reduces the cost of performing the verification.

In this paper, we make a step forward by adding a notion of *time* to our tool. Our temporal model, illustrated in Section 3, is in-

Sixth International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2007), September 3-4, 2007, Cavtat near Dubrovnik, Croatia.

Guarantee	Choices
Message Reliability	Absent, Present
Message Ordering	Random, Pair-wise FIFO, System-wide FIFO, Causal Order, Total Order
Filtering	Precise, Approximate
Subscription Propagation Delay	Absent, Present
Repliable Messages	Absent, Present
Message Priorities	Absent, Present, Present w/ Scrunching
Queue Drop Policy	None, Tail Drop, Priority Drop

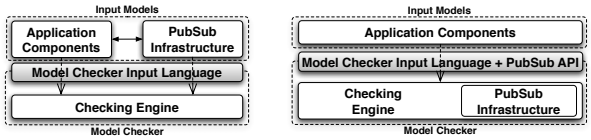
Table 1: P/S guarantees.

spired by the work on real-time event-based middleware by Deng et al. [10]. In their work, however, time was still tied to a particular incarnation of the P/S paradigm, namely the CORBA Event Service. Furthermore, they did not account explicitly for message delays, that may also impact the execution flow of a component. As such, their work cannot be reused as is. In our approach we bring time as an additional dimension next to those we use to characterize the semantics provided by P/S systems, explicitly accounting for message delays. By enabling the interplay between time and the various P/S guarantees, we enable the verification of P/S applications in *realistic* environments, going beyond the simplistic communication models of previous work. The effectiveness of our approach is assessed in a non-trivial case study, illustrated in Section 4, using properties expressed in Linear Temporal Logic (LTL).

To achieve our objective, we must delve into the internals of Bogor to modify critical aspects, such as the inter-component schedule. In doing so, we may run the risk of breaking the checking engine itself, thus producing unsound results. To address this issue, Section 5 illustrates how we formally verified the correctness of our temporal extension using existing tools for software verification. Notably, these are based on Bogor itself. This allowed us to make the verification process feasible, by leveraging off the expertise in Bogor we gained while developing the temporal extension. As we discuss in Section 5, a brute-force approach would instead make the same problem intractable. Brief concluding remarks and directions for future work conclude the paper in Section 6.

2. MODEL CHECKING P/S ARCHITECTURES

The P/S paradigm revolves around a few primitives, which allow application components to interact by publishing messages or issuing (un)subscriptions. Although the programming interface mostly remains the same across different P/S incarnations, the way the paradigm is implemented greatly differs. Table 1 illustrates a classification of P/S guarantees and semantics we found in existing systems. These characterize the features that may impact the behavior of components running on top of such infrastructures, and therefore affect whether a given requirement is actually verified. For instance, *message ordering* refers to the policy used to deliver messages: *random* order, *pair-wise FIFO* order to deliver messages to a given subscriber in FIFO order with respect to publish operations from the same component, *system-wide FIFO* order to deliver messages in the same order as publish operations also across different components, according to the *causality* chain among messages, or *total order* to deliver the same messages in the same order to all components with the same set of subscriptions. The remaining dimensions in Table 1 are thoroughly described in [2]. In the following, we briefly overview how the problem of verifying P/S architectures has been tackled in previous work, and how we addressed the same problem trough a domain-specific model checker.



(a) Standard. (b) Domain-Specific.

Figure 2: Approaches to model checking P/S architectures.

2.1 Approaches Using Standard Tools

Garlan et al. investigated the problem of model checking P/S architectures in [14]. They provide a set of pluggable modules that allow the user to choose one configuration out of a predefined set. Nonetheless, available models are far from fully capturing the different characteristics of existing P/S systems shown in Table 1. Also, application components cannot change their subscriptions at run-time. The same approach is extended in [5] by adding more expressive events, dynamic delivery policies and dynamic event-method bindings. Still, the dispatching mechanism is only characterized in terms of delivery policy (*asynchronous*, *synchronous*, *immediate* or *delayed*). Similarly, some of the authors of this paper addressed similar issues in [21] using the SPIN model checker [17]. The P/S infrastructure is characterized in terms of reliability, message delivery order, and subscription propagation delay. Therefore, several of the dimensions in Table 1 are still missing.

Techniques applicable to specific P/S systems have been considered in [3, 6]. Beek et al. [3] concentrate on the addition of a P/S notification service to an existing groupware protocol. They also show how the P/S paradigm improves the user awareness of the status of a project when used to coordinate a large development team. Caporuscio et al. [6] develop a compositional reasoning technique based on an assume-guarantee methodology. The methodology is applied on a specific case study, i.e., the development a file sharing system on top of the Siena P/S system [8]. These approaches lose generality in that they do not allow the user to customize the checking engine to model various P/S guarantees.

2.2 A Change of Perspective

Standard tools easily show their limitations when it comes to implement fine-grained, customizable models to describe guarantees such as those in Table 1. Essentially, this is due to the lack of parametrization in the input language, and state space explosion problems. Based on this observation, we reverted the traditional approach, by embedding the P/S communication paradigm *within* the model checker, and exporting the P/S API as primitive constructs of the modeling language. This is intuitively illustrated in Figure 2.

This approach provides several advantages over traditional solutions. We can easily customize the state space generation depending on the particular combination of guarantees assumed on the P/S infrastructure. This achieves a *domain-specific, state abstraction* mechanism that sensibly reduces the cost of accomplishing the verification by minimizing the number of states generated. By the same token, it is easy to customize the behavior of the P/S infrastructure. To this end, before starting the verification, the user selects a combination of the guarantees shown in Table 1. For instance, s/he may want to check the behavior of application components while assuming an underlying P/S infrastructure that guarantees causal order and precise filtering. Based on this, our tool instantiates a parametric dispatcher within the checking engine, to model the behavior the user desires. As a nice side-effect, describing the behavior of components running on top of a P/S infrastructure becomes straightforward, as the input language now comprises a set of constructs mimicking the P/S API found in real systems.

```
typealias MessagePriority int (0,9); enum DropPolicy {TAIL_DROP, PRIORITY_DROP}
extension PubSubConnection for polimi.bogor.bogorps.PubSubModule {
  typedef type<'a>;
  expdef PubSubConnection.type<'a> register<'a>();
  expdef PubSubConnection.type<'a> registerWithDropping<'a>(int, DropPolicy);
  actiondef subscribe<'a>(PubSubConnection.type<'a>, 'a -> boolean);
  actiondef publish<'a>(PubSubConnection.type<'a>, 'a);
  actiondef publishWithPriority<'a>(PubSubConnection.type<'a>, 'a, MessagePriority);
  expdef boolean waitingMessage<'a>(PubSubConnection.type<'a>);
  actiondef getNextMessage<'a>(PubSubConnection.type<'a>, lazy 'a);
}
```

Figure 3: Bogor preamble to export the P/S infrastructure.

```
// Message definition
record MyMessage { int value;}
MyMessage receivedEvent := new MyMessage;

// Subscription definition
fun isGreaterThanZero(MyMessage event)
    returns boolean = event.value > 0;

active thread PublisherComponent() {
  MyMessage publishedEvent;
  PubSubConnection.type<MyMessage> ps;

  loc loc0:  // Connection setup
    do {
      ps := PubSubConnection.register<MyMessage>();
    } goto loc1;

  loc loc1:  // Publishing a message
    do {
      publishedEvent := new MyMessage;
      publishedEvent.value := 1;
      PubSubConnection.
          publish<MyMessage>(ps, publishedEvent);
    } return;
}

active thread SubscriberComponent() {
  PubSubConnection.type<MyMessage> ps;

  loc loc0:  // Connection setup and subscription
    do {
      ps := PubSubConnection.register<MyMessage>();
      PubSubConnection.
          subscribe<MyMessage>(ps, isGreaterThanZero);
    } goto loc1;

  loc loc1:  // Message receive
    when PubSubConnection.
      waitingMessage<MyMessage>(ps) do {
      PubSubConnection.
          getNextMessage<MyMessage>(ps, receivedEvent);
    } return;
}
```

Figure 4: Using P/S extensions in a Bogor model.

To assess the feasibility of the approach, we use Bogor [19], an extensible model checker written in Java. With respect to similar tools, Bogor eases the definition of domain-specific constructs in its input language. Additionally, it provides out-of-the-box support for function pointers and dynamic threads, that are pivotal in modeling the dynamic applications we target. Adding further constructs to Bogor requires the developers to prepend a preamble to the Bogor models exploiting the new constructs, and provide one or more Java classes implementing the required semantics.

Figure 3 illustrates the preamble containing the P/S constructs available in our tool. Instead, Figure 4 shows an example use, where two components initially register with the P/S extension within Bogor, as shown in **`loc0:`**. In a sense, this models the operation of opening a connection to the P/S dispatcher, represented by a dedicated handler returned by the **`register`** operation. The handler is used to issue (un)subscriptions and publish messages over a specific connection to the dispatcher. The former is accomplished by providing as parameter to **`subscribe`** a *boolean function* representing the actual subscription, as in **`loc0:`** for the subscriber component. Notably, this gives the user great flexibility in defining the format of messages and the matching semantics. Instead, publishing is achieved with **`publish`** or **`publishWithPriority`**, depending on whether messages have associated priorities. To process incoming messages, a guard statement named **`waitingMessage`** is provided, which holds true when at least one message is available in the incoming queue. To retrieve the actual message content, we use the **`getNextMessage`** construct.

The mechanisms underpinning the constructs in Figure 3 focus on guaranteeing a given P/S semantics while reducing the number of states generated during the verification. For instance, consider the processing triggered by a message published: the dispatcher matches the message against the subscriptions issued so far, and delivers its content to a component if *at least one* of its subscriptions matches. However, in the presence of multiple subscriptions, the order in which these are examined is immaterial. Therefore, we can model the subscription table as a *set*, thus avoiding the generation of explicit states when it would make no difference from the application perspective. This already provides improvements over standard tools not equipped with a notion of set. Much greater improvements are achieved in controlling the generation of states representing message routing with particular delivery orderings, and in modeling message duplications. More details can be found in [2].

3. TIME EXTENSION

A large body of work exists in the field of model checking embedded systems with time constraints, e.g., [1]. However, our objective here is *not* to embed a generic notion of time. Being our approach specific to the P/S domain, we rather aim to include a temporal model suited to the requirements of applications built on top of a P/S infrastructure. Additionally, this must be sufficiently lightweight to enable its interplay with the other P/S dimensions in a clear and intuitive manner. Based on these reasons, we adapted the model presented in [10] to suit the aforementioned needs.

3.1 Time Model

Components running on top of P/S infrastructures are usually implemented as *passive* threads executing in an event-driven environment. Thread activation is accomplished implicitly by the P/S infrastructure in delivering a message to a subscribing component. This executes a message handler where further operations are usually performed, e.g., to issue new (un)subscriptions or publish messages. The thread is then suspended again waiting for further messages. In our approach, the *execution rate* of a component dictates the frequency of its operations on the P/S dispatcher, i.e., how many P/S primitives can be executed in a single time unit. Notably, a relevant class of real systems can be similarly modeled, e.g., [12,16]. In addition, in distributed environments it is unreasonable to assume that messages are delivered with zero delays. Therefore, differently

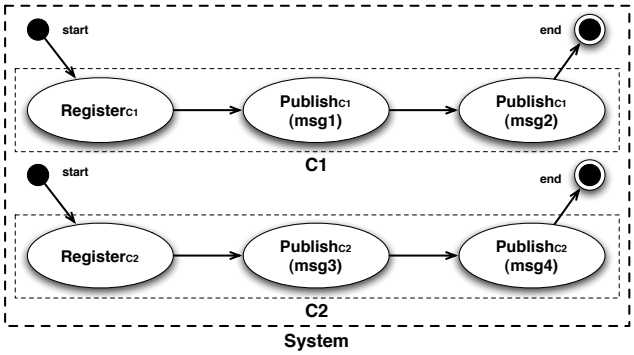
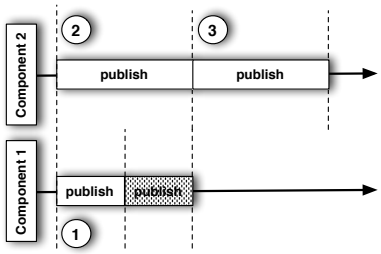
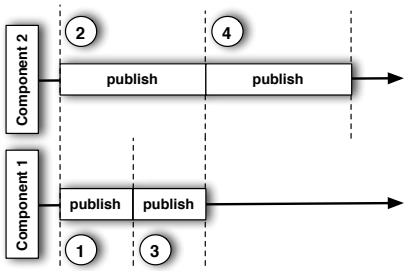


Figure 5: Two components publishing two messages each.



(a) Second execution: the time model is violated.



(b) Third execution: the timing constraints are met.

Figure 6: A graphical representation of the second and third executions in Table 2. (The numbers in the circles represent a possible system-wide schedule).

from [10], we also consider *random message delays*, thus providing an even more realistic environment to modeling P/S applications in case message delays are to be taken into account [7].

The above time model does *not* modify the individual states of the system. Rather, it *limits* the way the system state space is explored, by preventing some of the transitions to be taken. Let us consider the example in Figure 5: two components register with the P/S infrastructure, and publish two messages each. In the absence of any notion of time, our tool would explore all the possible interleavings of the operations of the two components. As the system global state is given by the combination of the per-component local states, the model checker would generate a high number of possible executions. Some example schedules are shown in Table 2.

When we enable our time model with component $C1$ being assigned an execution rate twice as that of $C2$, both executions 1 and 2 in Table 2 become unfeasible. The former trivially violates the timing constraints, as all operations of component $C2$ are executed before $C1$ starts. As for the latter, Figure 6 graphically compares execution 2 and 3: our time model is violated in transitioning from

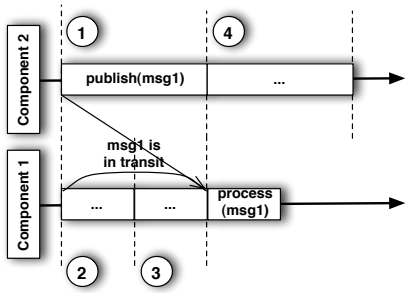


Figure 7: A possible schedule between two components when a message is sent with a non-zero delay.

$\langle publish(msg1)_{C1},\ publish(msg3)_{C2}\rangle$ to $\langle publish(msg1)_{C1},\ publish(msg4)_{C2}\rangle$. Indeed, being $C1$ running at twice the rate of $C2$, it should be allowed to perform two operations for each operation executed by $C2$. Instead, Figure 6(a) shows $C2$ performing a further operation while $C1$ has not yet performed the second publish. Also note that Figure 6(b) is not the only possible correct schedule. For instance, a different, correct execution is obtained by swapping the relative order of the initial publish operations. This indeed represents a different inter-leaving of concurrent operations.

Message delays are modeled similarly, by marking a message as "in transit" until the time constraints at the subscribing components are met. An example is illustrated in Figure 7: a component whose execution rate is of one operation per time unit publishes a message. This travels towards the subscribing components with a delay of a single time unit. The receiving component, whose execution rate is of two operations per time unit, has two available slots before the message appears in its input queue. During this time frame, it can either perform other operations, or decide to be suspended waiting for its input queue to fill.

3.2 Implementation

Implementing the above time model in our Bogor P/S extension essentially requires the ability to control the inter-component scheduling. To this end, we further augment the P/S preamble in Figure 3, by adding the constructs needed to control how the components proceed, and providing the corresponding semantics within the existing implementation of the P/S extension.

Bogor Language Constructs. Figure 8 illustrates the same example as in Figure 4, now using the additional constructs of our time extension. After registering the connection to the dispatcher, each component configures the time extension using **`configureTimeParams`**. This requires the component execution rate, and two values representing the lower and upper bounds of a (discrete) random delay for incoming messages. In case the user needs to temporarily revert to the original untimed behavior, it is sufficient to set to zero the execution rate of all components.

The inter-component schedule is controlled using two guard statements: **`canProceed`** and **`timedWaitingMessage`**. The former yields true when a component is allowed to proceed without violating any time constraint. Instead, the latter yields one value among **`CAN_PROCEED`**, **`CANNOT_PROCEED`**, and **`QUEUE_EMPTY`**. Note that we explicitly distinguish whether the component cannot proceed because higher priority components must be scheduled first (**`CANNOT_PROCEED`**), or the timing constraints would allow the component to proceed, but no messages are waiting in its input queue (**`QUEUE_EMPTY`**). The latter is needed to give the ability not to lose available slots in the current schedule and perform some operations instead of waiting for an incoming message, as it is possible in the untimed version of the tool.

Id	Execution
1	$\langle start_{C1}, start_{C2}\rangle, \langle start_{C1}, register_{C2}\rangle, \langle start_{C1}, publish(msg3)_{C2}\rangle, \langle start_{C1}, publish(msg4)_{C2}\rangle,$ $\langle start_{C1}, end_{C2}\rangle, \langle register_{C1}, end_{C2}\rangle, \langle publish(msg1)_{C1} end_{C2}\rangle, \langle publish(msg2)_{C1}, end_{C2}\rangle, \langle end_{C1}, end_{C2}\rangle$
2	$\langle start_{C1}, start_{C2}\rangle, \langle register_{C1}, start_{C2}\rangle, \langle register_{C1}, register_{C2}\rangle, \langle publish(msg1)_{C1}, register_{C2}\rangle,$ $\langle publish(msg1)_{C1}, publish(msg3)_{C2}\rangle, \langle publish(msg1)_{C1}, publish(msg4)_{C2}\rangle, \langle publish(msg2)_{C1}, publish(msg4)_{C2}\rangle,$ $\langle end_{C1}, publish(msg4)_{C2}\rangle, \langle end_{C1}, end_{C2}\rangle$
3	$\langle start_{C1}, start_{C2}\rangle, \langle register_{C1}, start_{C2}\rangle, \langle register_{C1}, register_{C2}\rangle, \langle publish(msg1)_{C1}, register_{C2}\rangle,$ $\langle publish(msg1)_{C1}, publish(msg3)_{C2}\rangle, \langle publish(msg2)_{C1}, publish(msg3)_{C2}\rangle, \langle publish(msg2)_{C1}, publish(msg4)_{C2}\rangle,$ $\langle end_{C1}, publish(msg4)_{C2}\rangle, \langle end_{C1}, end_{C2}\rangle$

Table 2: Some of the possible executions for the example in Figure 5, when no time notion is employed.

```
// Message definition
record MyMessage { int value;}
MyMessage receivedEvent := new MyMessage;

// Subscription definition
fun isGreaterThanZero(MyMessage event)
    returns boolean = event.value > 0;

active thread PublisherComponent() {
  MyMessage publishedEvent;
  PubSubConnection.type<MyMessage> ps;

  loc loc0:  // Connection setup
    do {
      ps := PubSubConnection.register<MyMessage>();
      PubSubConnection.configureTimeParams(ps, 2, 1, 0);
    } goto loc1;

  loc loc1:  // Publishing a message
    when (PubSubConnection.canProceed<MyMessage())
    do {
      publishedEvent := new MyMessage;
      publishedEvent.value := 1;
      PubSubConnection.
          publish<MyMessage>(ps, publishedEvent);
    } return;
}

active thread SubscriberComponent() {
  PubSubConnection.type<MyMessage> ps;

  loc loc0:  // Connection setup and subscription
    do {
      ps := PubSubConnection.register<MyMessage>();
      PubSubConnection.configureTimeParams(ps, 1, 1, 0);
      PubSubConnection.
          subscribe<MyMessage>(ps, isGreaterThanZero);
    } goto loc1;

  loc loc1:  // Message receive
    when (PubSubConnection.
      timedWaitingMessage<MyMessage>(ps)==CAN_PROCEED)
      do {
      PubSubConnection.
          getNextMessage<MyMessage>(ps, receivedEvent);
    } return;
    when (PubSubConnection.
      timedWaitingMessage<MyMessage>(ps)==QUEUE_EMPTY)
      do {
        // Do something...
    } return;
}
```

Figure 8: Adding time to the example model in Figure 4.

Bogor Internals. The mechanisms underlying the above Bogor constructs divide time into *frames*, whose length corresponds to a single operation of the lowest priority component. Based on this, higher priority components are scheduled multiple times in a single frame. Within a frame, all possible inter-leavings are generated. When this is achieved, the execution proceeds to the next frame.

Our implementation is tied to that of the P/S operations, to leverage off the *domain-specific semantics* of the executions involved, and cut down on the processing overhead whenever possible. Here we provide some examples as to where we take advantage of this. Interested readers are referred to [15] for more information.

- Several of the dimensions listed in Table 1 also somehow constrain the inter-component schedule. For instance, when *causal order* is assumed, a component is suspended waiting for incoming messages until all causally connected messages are in its input queue. In our experience, the impact of these guarantees on the number of enabled transitions is much greater than that imposed by the time model, especially when safety properties are to be checked. Therefore, whenever possible, we apply the mechanisms modeling the P/S guarantees *before* computing the time-based schedule and running the corresponding checks. This saves in the processing overhead during the verification.

- To model a random message delay between two bounds, we must generate all the possible executions corresponding to each (discrete) value in the interval. However, leveraging off the semantics of this value —which represents the time taken for a message to be transmitted from a component to another— it can be observed that not every value in the interval generates a different execution. Based on this observation, we can apply basic results of rate monotonic theory, and identify the subset of values that need to be checked to ensure the completeness of the verification. This way, we save the processing to generate executions that do not differ in the inter-leavings among components. Note that the above can be done while the verification proceeds, by looking at the execution rate and current state of the components about to receive the message in transit.

- When **`timedWaitingMessage`** returns **`QUEUE_EMPTY`**, the corresponding component already passed the time checks. Unless the component includes some alternative behaviors (as in Figure 8), it is suspended waiting for incoming messages. In this case, the checking engine lets another component proceed, and reschedules the suspended component immediately after, without re-running the time extension. Note that this is semantically correct because once a component passed the time checks for the current frame, there is no way for another component to subtract an allocated time slot from it. Based on this observation, we alleviate the processing overhead generated during the verification whenever a message is to be received.

Our tool performance is such that time-related properties can be checked within reasonable time under realistic assumptions on the P/S infrastructure, as illustrated next.

4. CASE STUDY

In this section, we describe the application scenario we have chosen to exemplify the approach, discuss the insights we gained by exploring the interplay between time and the various P/S guarantees, and report on some performance results assessing the effectiveness of our tool.

Scenario. Systems exploiting a P/S style of interaction span several applications domains. Among them, telemedicine is one of the most promising, as it has the potential to drastically lower the costs of maintaining hospital facilities, while letting patients enjoy better quality of life [20]. Here we consider a remote patient monitoring system, consisting of the following components:

- A variable number of *patients*, equipped with several sensing devices to monitor critical parameters, such as heart rate or blood pressure.
- The *medical laboratory*, responsible for monitoring the patients' status. In case of moderate danger, the lab personnel can immediately decide on corrective actions when no physical intervention is required. For instance, a dose change for a treatment can be remotely communicated to the patient.
- If the patient is in severe danger and is to be picked up by a first-response team, the medical laboratory informs a *flying squad* about the emergency, communicating all the relevant information to reach the patient and cope with the situation.
- In the same conditions, the medical laboratory also notifies the *hospital* about a possible request for hospitalization. On the way to it, the flying squad also keeps the hospital posted about the patient's current conditions, until a final notification is sent when the patient is handed over to the hospital personnel.

Interactions are expressed in terms of P/S operations. Specifically, the medical laboratory issues a subscription to collect the data gathered by the patient's sensors when the values are outside the allowed ranges. The hospital, as well as the flying squad, subscribe to all messages regarding possible requests for hospitalization. In addition, the hospital is also interested in messages coming from the flying squad while carrying a patient.

Essentially, three patterns of interactions characterize our scenario, depending on the patient's status. Under *normal conditions*, the component modeling the patient periodically publishes messages that, by virtue of not representing any possible danger, are not delivered to any remote component. When the patient parameters represent a *moderate danger*, the medical laboratory interacts with the patient only, e.g., to adjust the doses, without involving any further component. Differently, under *severe danger*, all the components in the system are involved until the patient's responsibility is passed to the hospital personnel.

Running the Verification. To verify our initial design, we checked whether the following requirements were satisfied:

Requirement 1 When a patient's status turns into a situation of moderate danger, any corrective action must be communicated by the medical laboratory within $T1$ time units.

Requirement 2 Whenever a patient is in a situation of severe danger, the hospital must receive a request for hospitalization within $T2$ time units.

Requirement 3 When a patient arrives at the hospital, the personnel there must have received the corresponding request for hospitalization in advance.

The above requirements are straightforwardly expressed as LTL formulae over the variables representing the components' current state[1]. In particular, the first and second requirement exploit a hook into our time extension that allows properties to be expressed depending on time intervals.

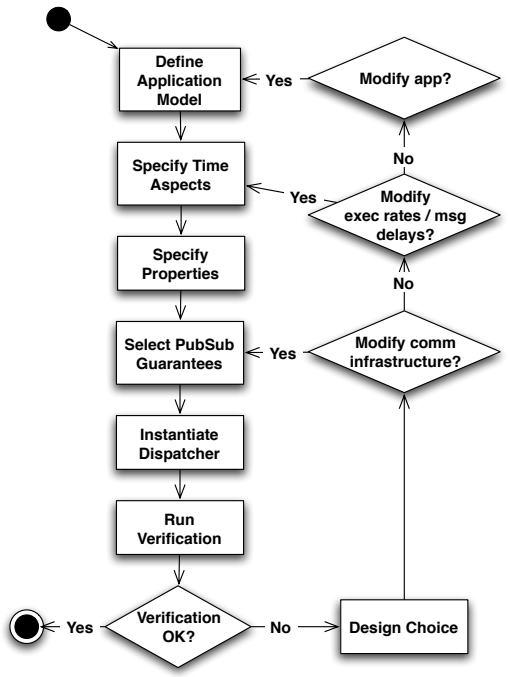


Figure 9: Verification flow with Bogor and our P/S extension.

As illustrated in Figure 9, our tool allows the application designer to iterate in a loop where either the application model evolves, the timing aspects are tuned, or the guarantees assumed on the P/S infrastructure change. This allows the designers to explore the interplay between the application and the underlying communication infrastructure. Further, time adds another dimension to this, enabling an additional degree of freedom.

For instance, we realized that in our application the characteristics of the input queues and the component execution rates are tightly intertwined. Indeed, the first requirement easily fails if the component modeling the medical laboratory is not assigned an execution rate sufficient to handle multiple concurrent notifications coming from different patients. However, even if this component is running at a sufficiently high rate, the patients' notifications can easily fill up the component's input queue if this is assumed to be finite. In this case, depending on the drop policy adopted, some messages are discarded. Similarly, when multiple patients are in severe danger, the medical laboratory may send multiple requests for hospitalization. Therefore, to meet the second requirement, the component modeling the hospital must be able to process these messages within a given time bound, and have sufficiently large queues not to drop any of them.

An interesting relation also exists between message delays and delivery order. To verify the third requirement, our application needs an underlying communication infrastructure providing causal order delivery. Indeed, with random message delays, it may happen that the message coming from the medical lab is delayed w.r.t. the one sent by the flying squad when handing over the patient. However, if message delays are constant, the system essentially proceeds in a *delayed-synchronous* manner, which makes assuming any specific message orderings superfluous.

In addition, to evaluate the performance of our tool, we measured the *time* and *memory* taken, as well as the *number of states*

[1]To run LTL verification, we used the Bogor extension in [4]

Req.	No. Patients	Mem. (Mb)	States	Time
R1	10	278.38	70234	$\approx$ 16 min
R1	20	312.31	123122	$\approx$ 20 min
R2	10	412.21	113213	$\approx$ 22 min
R2	20	502.75	209123	$\approx$ 26 min
R3	10	498.1	232123	$\approx$ 30 min
R3	20	591.1	289124	$\approx$ 35 min

Table 3: Performance of our tool when R1-R3 are verified.

generated during the verification. We considered a system with 10 or 20 patients, each publishing 10 messages that may randomly trigger the actions corresponding to moderate or severe danger. We gathered the aforementioned metrics on an Intel Core Duo 1,83Ghz processor running Apple OSx, using the DJProf [11] profiler to evaluate the memory occupancy.

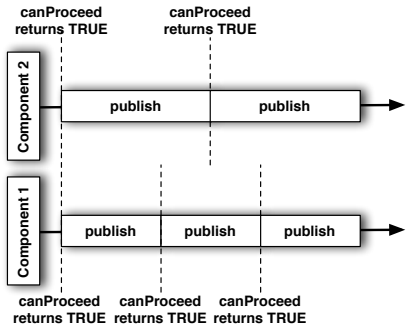
When a requirement turns out not to be verified, a counterexample is returned within a few seconds. Instead, Table 3 reports the performance of our tool in case the verification succeeds. Note that *doubling* the number of patients corresponds to a sharp increase in the message traffic modeling the interactions among components, as well as in the number of possible inter-leavings. The additional complexity of the model, however, yields only a *moderate* overhead in the time taken to accomplish the verification, about 25% in the worst case. We believe this is due to our implementation of the time extension, described in Section 3.2, that exploits the domain-specific semantics of P/S to cut down the processing. By the same token, the above metrics only slightly change assuming different P/S guarantees that still make the verification succeed. For instance, as already mentioned, the third requirement can be verified by either assuming causal ordering, or by imposing a constant message delay. In both cases, the verification completes in about half hour.

5. VERIFYING THE TIME EXTENSION

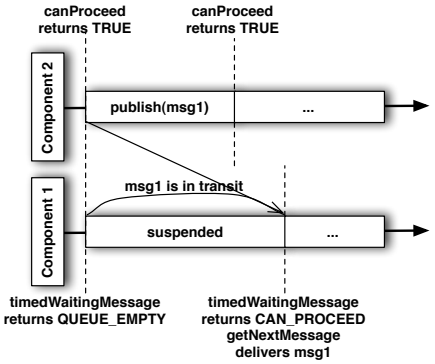
For our tool to prove useful, we must provide a strong foundation upon which our implementation can substantiate the soundness of presented results. In general, it is hard to achieve this objective. Moreover, in our approach the challenge is even more critical, as we are extending an existing model checker by augmenting its internal mechanism. To address this issue, we checked the correctness of our temporal extension using Bandera [9], a tool for the automatic verification of Java code. Notably, Bandera itself is based on Bogor. Essentially, it translates the Java code into a Bogor model upon which the actual verification is run. Consequently, we exploited our expertise in Bogor to reduce the size of the code fed as input to Bandera, therefore making the verification feasible. In this section we report on our experience in this respect, highlighting the lessons we learned on the way.

Bandera is a set of tools for the transformation of Java code into verifiable models. To this end, code analysis techniques are used to reduce the size of the models produced. Essentially, Bandera aims to i) eliminate from the input code all the elements (e.g., classes, methods, variables) that do not affect the verification of a given property, ii) abstract the type of, and infer bounds for, the remaining variables to reduce the state space generated during the verification. This is achieved through multiple translation steps, whose final output is a runnable Bogor model encompassing the properties to be verified. These are generally specified in terms of the values taken by input or output parameters of relevant methods.

Despite the degree of sophistication of Bandera, a brute-force approach whereby the entire Bogor code plus the P/S and time extensions are input to Bandera easily fails: the model output by Bandera is intractable. Nonetheless, by examining the outcome of



(a) Two components publishing messages at different rates.



(b) A subscriber and a publisher component, messages have a non-zero delay.

Figure 10: Scenarios for verifying the time extension.

Bandera, one can recognize how large parts of those models are not relevant to the verification of the time extension. Indeed, as we already observed, our notion of time does not alter the state space w.r.t. the untimed version of our tool. Rather, it limits the way the state space is explored. Therefore, the correctness of our extension can be checked by simply making sure that the guards controlling the inter-component schedules return the correct values for every possible situation.

Based on the above observation —that again exploits our domain-specific knowledge— we carved out the time extension plus a few Bogor components needed to trigger its functionality. Specifically, almost the entire Bogor code enabling the extension capabilities was removed, as well as parts of the Bogor parser. Moreover, the state space generation mechanism was greatly reduced, as only the ability of *generating* the state space was required. Instead, how to *explore* this is dictated by the time extension, that is our verification target. To let the entire package compile, we implemented empty stubs in place of the parts removed.

To check our implementation, we must explore all the possible inter-component schedules. Notably, this can be accomplished with only two components, and four scenarios where these components publish or receive messages:

Scenario 1. As shown in Figure 10(a), two components publish messages with a non-integer ratio between their execution rates. No component is subscribed to these messages, hence they are discarded at the dispatcher. The scenario essentially checks whether the inter-component schedules are generated correctly in the absence of any message in transit.

Scenario 2. With a non-zero message delay, a component subscribes

to a message published by another component, as depicted in Figure 10(b). Therefore, **`timedWaitingMessage`** returns **`QUEUE_EMPTY`** while the message is in transit, and switches to **`CAN_PROCEED`** as soon as the message appears in the input queue. The component execution rates are assigned so that only the receiving component is allowed to proceed at the time of message reception. The scenario verifies the functioning of **`timedWaitingMessage`**, and how the inter-component schedule is generated when a message is received with the subscribing component being given higher priority.

Scenario 3. Similarly to the previous scenario, this time the component execution rates are assigned so that the publishing component has higher priority. Therefore, when the message is inserted in the input queue of the receiving component, this is not immediately scheduled, and the publishing component can proceed. This scenario checks the situation dual to scenario 2.

Scenario 4. To test the combination of scenarios 2 and 3, the component execution rates and message delays are assigned so that *both* components can be scheduled when the message arrives at the intended recipient. The objective is to check whether both possible schedules are correctly generated.

Overall, Bandera generated about 100 assertions to verify the correctness of our implementation. As for the results, we actually discovered a bug in our initial prototype. Bandera showed a counterexample in the third scenario where **`timedWaitingMessage`** returned the wrong value after backtracking from the state that represents component A receiving the message. This was caused by a non-initialized variable, whose default value worked for most (but not all) combinations of the input parameters. Apparently, in our initial tests we were lucky in picking the "right" inputs. This result witnesses the importance of our efforts in verifying our time extension. In their absence, this bug would have probably survived.

6. CONCLUSION

In this paper we presented a time model to investigate time-sensitive P/S architectures. In our approach, time is embedded within the model checker as an additional dimension characterizing the system behavior. This work completes previous efforts by some of the authors, by providing the missing tile in a framework for the verification of P/S architectures. Our approach opens up opportunities for better investigations during the early design stages, which ultimately hold the potential to produce more reliable implementations.

Our research agenda includes a deeper assessment of the effectiveness of our approach through several case studies, as well as further work in the direction of the formal verification of the correctness of our Bogor implementation.

7. REFERENCES

[1] R. Alur, C. Courcoubetis, and D. Dill. Model-checking for real-time systems. In *Proc. of the* 5^{th} *Int. Symposium on Logic in Computer Science*, 1990.

[2] L. Baresi, C. Ghezzi, and L. Mottola. On accurate automatic verification of publish-subscribe architectures. In *Proc. of the* 29^{th} *Int. Conf. on Software Engineering (ICSE07)*, 2007.

[3] M.-H. Beek, M. Massink, D. Latella, S. Gnesi, A. Forghieri, and M. Sebastianis. A case study on the automated verification of groupware protocols. In *Proc. of the* 27^{th} *Int. Conf. on Software engineering (ICSE05)*, 2005.

[4] Bogor Extensions for LTL Checking. `projects.cis.ksu.edu/projects/gudangbogor/`.

[5] J.-S. Bradbury and J. Dingel. Evaluating and improving the automatic analysis of implicit invocation systems. In *Proc. of the* 9^{th} *European software engineering Conf.*, 2003.

[6] M. Caporuscio, P. Inverardi, and P. Pelliccione. Compositional verification of middleware-based software architecture descriptions. In *Proc. of the* 26^{th} *Int. Conf. on Software Engineering (ICSE04)*, 2004.

[7] N. Carvalho, F. Araujo, and L. Rodrigues. Reducing latency in rendezvous-based publish-subscribe systems for wireless ad hoc networks. In *Proc. of the* 5^{th} *Int. Wkshp. on Distributed Event-Based Systems (DEBS)*, 2006.

[8] A. Carzaniga, D.-S. Rosenblum, and A.-L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Trans. Comput. Syst.*, 19(3), 2001.

[9] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zheng. Bandera: extracting finite-state models from java source code. In *Proc. of the* 22^{nd} *Int. Conf. on Software engineering*, 2000.

[10] X. Deng, M.-B. Dwyer, J. Hatcliff, and G. Jung. Model-checking middleware-based event-driven real-time embedded software. In *Proc. of the* 1^{st} *Int. Symposium on Formal Methods for Components and Objects*, 2002.

[11] DJProf Java Profiler, `www.mcs.vuw.ac.nz/djp/djprof/`.

[12] B. S. Doerr and D. C. Sharp. Freeing product line architectures from execution dependencies. In *Proc. of the* 1^{st} *Conf. on Software product lines : experience and research directions*, 2000.

[13] P.-Th. Eugster, P.-A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2), 2003.

[14] D. Garlan and S. Khersonsky. Model checking implicit-invocation systems. In *Proc. of the* 10^{th} *Int. Workshop on Software Specification and Design*, 2000.

[15] G. Gerosa. Design and Implementation of a Time Extension for a Domain-Specific Model Checker. Master Thesis (in italian), Politecnico di Milano (Italy), 2007.

[16] T. H. Harrison, D. L. Levine, and D. C. Schmidt. The design and performance of a real-time corba event service. In *Proc. of the* 12^{th} *ACM SIGPLAN Conf. on Object-oriented programming, systems, languages, and applications (OOPSLA)*, 1997.

[17] G. J. Holzmann. The model checker spin. *IEEE Trans. Softw. Eng.*, 23(5):279–295, 1997.

[18] S. Li, Y. Lin, S.H. Son, J.A. Stankovic, and Y. Wei. Event detection services using data service middleware in distributed sensor networks. *Telecommunication Systems*, 26(2), June 2004.

[19] Robby, M.-B. Dwyer, and J. Hatcliff. Bogor: an extensible and highly-modular software model checking framework. In *Proc. of the* 9^{th} *European software engineering Conf.*, 2003.

[20] U. Varshney. Pervasive healthcare. *Computer*, 36(12), 2003.

[21] L. Zanolin, C. Ghezzi, and L. Baresi. An approach to model and validate publish/subscribe architectures. In *Proc. of the SAVCBS Workshop*, 2003.

On Timed Components and their Abstraction

Ramzi Ben Salah
VERIMAG
2, av. de Vignate
38610 Gieres, France
Ramzi.Salah@imag.fr

Marius Bozga
VERIMAG
2, av. de Vignate
38610 Gieres, France
Marius.Bozga@imag.fr

Oded Maler
VERIMAG
2, av. de Vignate
38610 Gieres, France
Oded.Maler@imag.fr

ABSTRACT

We develop a new technique for generating *small-complexity abstractions* of timed automata that provide an approximation of their *timed input-output behavior*. This abstraction is obtained by first augmenting the automaton with additional *input clocks*, computing the "reachable" timed automaton that corresponds to the augmented model and finally "hiding" the internal variables and clocks of the system. As a result we obtain a timed automaton that does not allow any qualitative behavior which is infeasible due to timing constraints, and which maintains a relaxed form of the timing constraints associated with the feasible behaviors. We have implemented this technique and applied it to several examples from different application domains.

Categories and Subject Descriptors

D.2.13 [**Software Engineering**]: Reusable Software—*Reuse models*

General Terms

Verification

Keywords

components, timed automata, abstraction

1. INTRODUCTION

The basic premise of a component-based design methodology is that a component (a hardware IP block, a software module, a network router) can be used during the construction of a system without deep knowledge of its intimate internal structure but rather using a more abstract (and conservative) description of its observable input-output behavior. This description should be sufficiently detailed to prove the correct interaction of the component with the entire system, and sufficiently small to avoid state explosion. In this work we extend this methodology to *timed systems* models that reflect also *quantitative performance* information. Phenomena such as delays in circuits and communication networks, as well as execution and response times of software are the natural application domains for such models. Using the new abstraction technique presented in the paper, we can *automatically* build a conservative approximation of the *timed input-output behavior* of the component such that any performance guarantees obtained using the abstract model, hold also for the concrete model.

This technique, which has been implemented into a tool, transforms a high-level description of the timed systems[1] into a product of timed automata that captures all the possible behaviors of the system under *all admissible inputs* and *choices of delay parameters*. From this automaton which has one state variable and one clock variablefor every timed element[2] we generate an abstract model with fewer states and clocks which provides an *over-approximation of the time-dependent input-output behavior* of the system. This simplified model can replace the original model within a hierarchical/compositional reasoning methodology. Our technique allows the user to select the appropriate level of aggressiveness in the abstraction, that is, the level of relaxation of the timing and ordering constraints in the abstract model, to achieve a good trade off between the complexity of the model and its faithfulness to the concrete behavior of the system. The major steps in our procedure are:

1. Introduction of additional *input clocks*, each of which measures the time elapsed since the occurrence of a particular input event. When the effect of this event is propagated through the system, its associated clock is deactivated and can be reused by future events.[3] These "dynamic clocks" constitute a novel and non-trivial feature in the theory and practice of timed automata and their number is always bounded, depending on the variability of the input and the structure of the system.

2. Full-fledged reachability analysis of the automaton, resulting in a modified automaton from which all behaviors that violate timing constraints are eliminated.

3. Generation of an abstract model by *hiding* all internal clocks and variables and *projecting* the timing constraints on the input clocks.

[1]For circuits this description consists of a network of logical gates with bi-bounded delay elements, for embedded software it consists of descriptions of tasks, resources, durations and scheduling policies.

[2]A timed element is something that measures the time since the occurrence of some event and uses this value to guard a transition.

[3]We restrict ourselves to systems with an *acyclic* structure, systems in which every cycle in the transition graph has at least one transition labelled with an input event. Such systems do not generate "autonomous" cycles and hence every input event generates a "wave" of reactions that propagate through the system within a finite time.

Sixth International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2007), September 3-4, 2007, Cavtat near Dubrovnik, Croatia.

4. Minimization of the automaton by merging states which are equivalent (or approximately-equivalent) with respect to observable input-output behavior.

The rest of the paper is organized as follows. Section 2 gives some background on abstraction in general while Section 3 offers a quick survey of timed automata and the computational difficulty inherent in their analysis. Section 4 illustrates our modeling approach and describes the various stages of our abstraction procedure. Preliminary experimental results are described and discussed in Section 5 followed by suggestions for future work.

2. ABSTRACTION IN GENERAL

In verification and other system design activities we have often to deal with a system model S which is too complex to analyze due to its large or even infinite state space. In this case we can try to replace S with a more abstract model S' with the following properties: 1) The complexity of S' is smaller than that of S, where complexity is viewed operationally, that is, S' is easier to analyze than S using some verification tool; 2) Every observable behavior of S is also a behavior of S', but not vice versa (conservative approximation).

Analyzing S' is computationally easier than the verification of S but due to over approximation, it may happen that the verification of S' may fail although S is correct. The navigation in the space of possible abstractions of S in order to find one which is sufficiently simple to avoid explosion yet sufficiently detailed to prove the property in question, is a major research topic, especially for infinite-state systems such as those used to model software. The current paper is concerned with adapting this methodology to *timed systems* defined using the *timed automaton* formalism, but before moving to those, let us contemplate briefly on the nature of abstraction.

A discrete component S, such as a digital circuit or a reactive program, is a device that maintains some relationship between the sequence of inputs it observes and the sequence of outputs it emits. Mathematically speaking, it can be viewed as a *transducer*, an input-output transition system $S = (X, Y, Z, \delta, \gamma)$ that reads inputs ranging over X, makes transitions in its state space Y, according to the transition relation δ, and outputs elements of Z according to the output function γ. If we view S as a "white box" and observe also the sequence of states visited while producing the output (see Figure 1-(a)), we can view S as realizing some sequential function f from X^* to $Y^* \times Z^*$ However, we do not really care about the internal states of S, it is only the input-output function (or relation) from X^* to Z^* which determines whether S interacts correctly with its environment and meets its specification. So the most natural simplification is to hide Y and consider the sequential function $f : X^* \to Z^*$ as the essence of S.

However, contrary to what one may prematurely think, hiding Y and projecting onto the output does not imply that we gain anything in complexity neither lose anything in accuracy. The reason is that every sequential function has its *inherent state space structure* (minimal realization, Myhill-Nerode congruence relation), regardless of whether the states themselves are observable. In other words, hiding internal states from outside observation does not change the state space nor the transition function, which remains of the form $y' = \delta(y, x)$. The only thing it does is to "remove" the states from the output function (see Figure 1-(a)).

Real abstractions, do reduce complexity and lose information by simplifying the transition relation. The most common way to do so is to define an equivalence relation $\sim$ on Y and replace $S = (X, Y, Z, \delta, \gamma)$ with $S' = (X, Y', Z, \delta', \gamma')$ where $Y' = Y/\sim$, the set of partition blocks of $\sim$. In other words we merge together states that are $\sim$-equivalent (see Figure 1-(b)). A transition (y'_1, x, z, y'_2) exists in S' if a transition (y_1, x, z, y_2) exists for *some* $y_1 \in y'_1$ and $y_2 \in y'_2$. Such an abstraction may lose information and generate more behaviors then are really possible in S. For example, the behavior x_1/z_1z_4 is possible in S' while in S we have either x_1/z_1z_3 or x_2/z_2z_4.

Our goal is to export these ideas to timed automata by hiding some clocks variables, but the explanation is more complicated because in timed automata, like in any other automata with auxiliary variables, the visible transition graph does not convey all the information on the system dynamics but rather a projection of it.

3. TIMED MODELS

Timed extensions of discrete transition systems, such as timed automata or Petri nets, allow one to reason about systems in an *extremely important level of abstraction*. At this level, the process of switching between two discrete states is refined into two transitions, *initiation* and *conclusion*, separated by some real-valued *delay*, which is often not known exactly but bounded. Among the numerous phenomena that can benefit from this style of modeling we mention the execution time of a block of code in a real-time program, communication delays in a network, and the time it takes for a digital electronic gate (or a more complex block) to stabilize to a new value after its input has changed. In this paper we demonstrate our approach using models based on networks of Boolean gates due to their notational economy and because it is easy to generate large examples in a uniform way, but the techniques developed can be adapted to other description levels and application domains.

Timed automata model *duration* of actions using auxiliary *clock variables*. To express a timing constraint between two transitions (such as the initiation and termination of a process) a clock is reset to zero while taking the first transition, and its value is tested to satisfy the constraint as a pre-condition ("guard") for the second transition. Between transitions, when the automaton stays in a state, the value of all active clocks progress in the same pace, representing at each moment the time elapsed since the occurrence of their respective events.

At each moment along the real-time axis, the state of the automaton is characterized by a configuration (q, v) with q being a discrete state and v a vector of clock valuations ranging over some bounded subset of $\mathbb{R}^n$. Albeit the infinite state space, the basic verification questions for timed automata are decidable [1]. Existing decision techniques suffer, however, from the usual state-explosion problem, aggravated by the clock-explosion problem: during reachability analysis we need to store "symbolic states" of the form (q, P) where q is a discrete state and P is a *set* of clock valuations. These sets are expressed by a conjunction of constraints of forms like $x < d$ or $x - y < d$, and constitute a special class of convex polyhedra that we call *timed polyhedra*. In a state where n clocks are active, timed polyhedra can be n-dimensional and admit up to two constraints for each pair of variables. Consequently the analysis of a system consisting of n timed components may generate in the worst case $O(2^n \cdot n!)$ symbolic states, each with an $O(n^2)$ representation size. Although a lot of effort has been invested during the last decade in finding more efficient ways to analyze timed automata, scalability toward the size requirements of circuit analysis has not been achieved. In this work we start exploring *compositional reasoning* via *abstraction* as an alternative road toward scaling-up timed automata technology.

4. TIMED ABSTRACTION

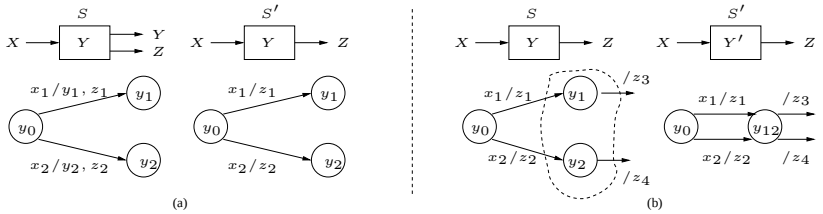


Figure 1: (a) Hiding internal states from the outside does not necessarily reduce complexity; (b) An abstraction S' of S obtained by merging y_1 and y_2 into y_{12}.

Timed automata are quite intuitive but their formal definition can be rather irritating outside formal verification circles. To address potential users of the proposed technology we avoid formalization and illustrate our technique using a running example.

4.1 Modeling

Figure 2 shows a *timed Boolean circuit* and one of its possible behaviors. The circuit has an input signal x which may switch arbitrarily, but with bounded variability, that is, it has to wait at least 5 time units between subsequent switchings. Changes in x are propagated through a *bi-bounded delay element* whose output y follows the value of x within some $t \in [1, 2]$ time units and is fed into a similar delay element with output z. Mathematically speaking the relation between signals maintained by the circuit can be expressed by the *delay inclusions* $y \in D_{[1,2]}(x)$ and $z \in D_{[1,2]}(y)$

Following the principles laid out in [7] we model the input and the components using the automata $\mathcal{A}_x$, $\mathcal{A}_y$ and $\mathcal{A}_z$ of Figure 3. We label transitions by *input events, guards, clock resets* and *output events*, for instance, a transition labeled by $x^+, c_y < 2/\{c_z\}, y^-$ can be taken upon the rising of x, provided that $c_y < 2$, and its effect is to reset c_z and lower y. The input automaton $\mathcal{A}_x$ guarantees bounded variability by guarding its transitions with the condition $c_x > 5$ and by resetting clock c_x to zero at every transition.

The modeling of *delay elements* by *timed automata* is a crucial ingredient of our methodology. The automaton $\mathcal{A}_y$ starts at a stable state 0 where its value coincides with the value of its input x. Upon a change in x it moves to an *excited state* $0'$ while resetting its clock c_y. The "stabilize" transition from $0'$ to 1 through which y "catches up" with x, may happen when $c_y \in [1, 2]$, that is, inside the time window $[t + 1, t + 2]$ with t being the time when x has changed.[4] Note that the "excite" transition from 0 to $0'$ is always triggered by an external input but is not visible from the outside, while the stabilization transition from $0'$ to 1 is generated autonomously without an input event (unless one considers the passage of time as such) and is visible to the outside world. Composing the three automata we get the global automaton $\mathcal{A}$ of Figure 4. Note also that each clock is *active* only in global states in which its corresponding gate is excited.

There are different approaches for treating the case where x changes its value again *before propagating to* y. For the purpose of this work, we assume that the automaton returns from $0'$ to 0 (a "regret" transition) and thus it "forgets" the whole episode. Other approaches may treat this phenomenon as an error ("glitch"), or model it in a manner more faithful to the physical realization of logical gates. Either way, this guarantees that the number of events that may be "alive" in the systems is bounded, regardless of the input frequency. In other domains this effect can be achieved by admission controllers or bounded buffers.

The *semantics* of this automaton consists of all xyz signals it can generate, that is, the signals carried by all runs of the automaton. These runs are sequences of configurations separated by transitions or by time-passage periods. The behavior where x rises at 6, y follows after 1 time unit and z follows 1.9 time units after y, is captured by the following run where configurations are presented as tuples of the form

$$\begin{pmatrix} x, c_x \\ y, c_y \\ z, c_z \end{pmatrix}$$

where $\perp$ denotes inactive clocks:

$$\begin{pmatrix} 0', 0 \\ 0, \perp \\ 0, \perp \end{pmatrix} \xrightarrow{6} \begin{pmatrix} 0', 6 \\ 0, \perp \\ 0, \perp \end{pmatrix} \xrightarrow{x^+} \begin{pmatrix} 1', 0 \\ 0', 0 \\ 0, \perp \end{pmatrix} \xrightarrow{1} \begin{pmatrix} 1', 1 \\ 0', 1 \\ 0, \perp \end{pmatrix}$$

$$\xrightarrow{y^+} \begin{pmatrix} 1', 1 \\ 1, \perp \\ 0', 0 \end{pmatrix} \xrightarrow{1.9} \begin{pmatrix} 1', 2.9 \\ 1, \perp \\ 0', 1.9 \end{pmatrix} \xrightarrow{z^+} \begin{pmatrix} 1', 2.9 \\ 1, \perp \\ 1, \perp \end{pmatrix}$$

The circuit behavior carried by this run can be represented either in a state-based manner (as a signal) or in an event-based manner (as a time-event sequence, see [2]) as follows:

$$\begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}^{6} \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}^{1} \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix}^{1.9} \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} \quad 6 \cdot x^+ \cdot 1 \cdot y^+ \cdot 1.9 \cdot z^+$$

We use the term *qualitative behavior* to denote the sequence of signal values without reference to timing. For this example the qualitative behavior is $x^+y^+z^+$ and can be viewed as an equivalence class of all signals of the form $t_1 \cdot x^+ \cdot t_2 \cdot y^+ \cdot t_3 \cdot z^+ \cdot t_4$ for any $t_1, t_2, t_3, t_4 \geq 0$.

If we ignore timing constraints, remove all references to clocks from transition guards and leave only the rising and falling labels, we obtain a timed automaton which is practically equivalent to an untimed automaton. This can be viewed as a very aggressive form of abstraction whose set of qualitative behaviors is the set of all sequences of labels carried by *all paths* in the transition graph, for example $x^+y^+z^+x^-y^-z^-$ or $x^+y^+x^-y^-$. However, taking timing into account one can see that given the variability constraint on x, the second behavior is impossible because state $110'$ is never reached with a combination of clock values that satisfies the guard $c_x > 5 \wedge c_z < 2$.

4.2 Reachability Analysis

The analysis of the timed automaton itself, rather than its untimed abstraction, is typically performed by constructing the *reachability graph*, also known as the *simulation graph* [6], which gives

[4]The fact that the automaton *must* leave state $0'$ when c_x reaches 2 can be expressed either using staying conditions ("invariants") associated with states, or "deadlines" and "urgencies" associated with transitions, [9]. Using the latter terminology, stabilization transitions are *delayable*.

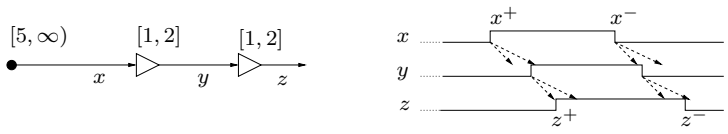


Figure 2: A simple timed circuit and a typical behavior.

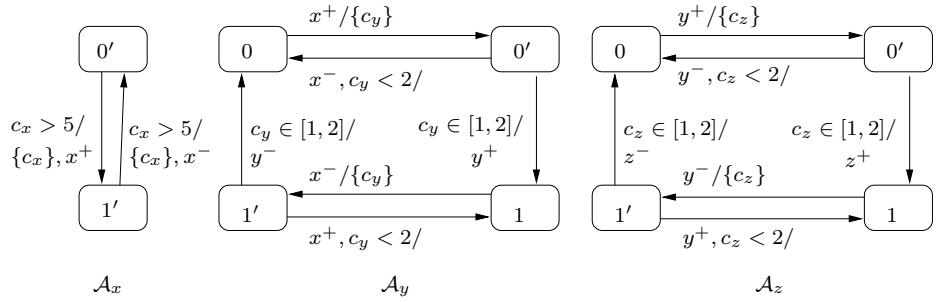


Figure 3: Modeling the circuit of Figure 2 with timed automata.

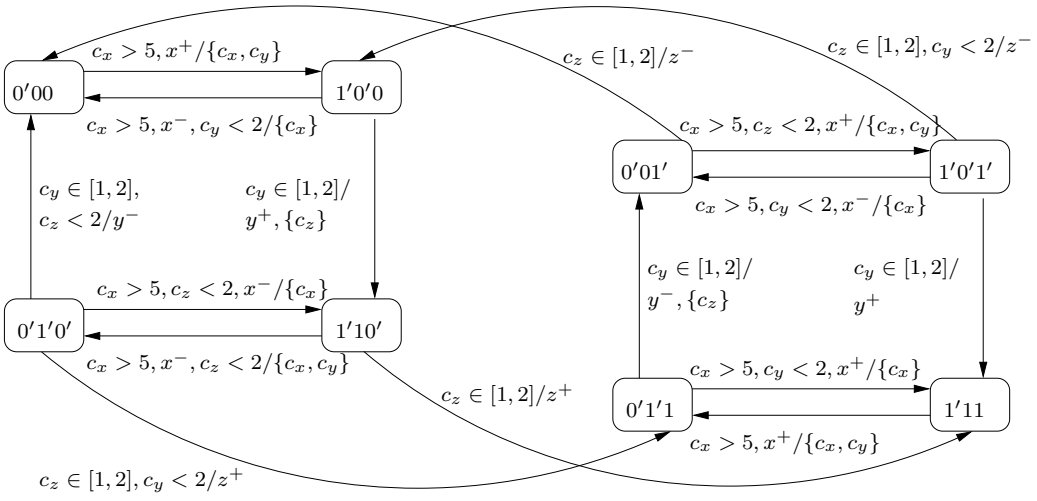


Figure 4: The global automaton $\mathcal{A} = \mathcal{A}_x \circ \mathcal{A}_x \circ \mathcal{A}_z$ for the circuit.

a (somewhat non-intuitive) representation of that part of the timed automaton which is reachable from some initial state or set of states. To illustrate the idea let us compute the reachability graph for $\mathcal{A}$ starting from an initial configuration $(0'00, c_x = 0)$. In this state we can let time progress indefinitely and can reach all clock valuations satisfying $c_x \geq 0$. This is represented as a "symbolic state" $(0'00, c_x \geq 0)$. The next step is to intersect this set with the transition guard $c_x > 5$ to obtain all the configurations from which the transition labeled by x^+ can be taken, represented by the symbolic state $(000, c_x \geq 5)$. Finally, by applying to this set the resetting of c_x and c_y, we obtain the symbolic state $(10'0, c_x = c_y = 0)$.

The process is then repeated from the new symbolic state where time passage is limited by 2 which is the upper bound on the rising of y, hence the symbolic state is $(10'0, c_x = c_y \leq 2)$. The transition back to $0'00$ cannot be taken due to empty intersection with the guard $c_x > 5$ and this transition is eliminated. The intersection with the guard $c_y \in [1, 2]$ gives $(10'0, 1 \leq c_x = c_y \leq 2)$ and the result of the transition after resetting c_z is $(1'10', c_x \leq 2 \wedge c_z = 0)$. In this state time can progress until $c_z = 2$ resulting in the symbolic state $(1'10', 1 \leq c_x - c_z \leq 2 \wedge c_z \leq 2)$ and so on and so forth until we obtain the reachability graph of Figure 5. The procedure is guaranteed to terminate due to the finite number of bounded timed polyhedra [1, 6].

We interpret the reachability graph as a timed automaton $\mathcal{A}'$ as follows: for each symbolic state (q, P) we define a copy of state q whose staying condition (and its outgoing transition guards) are restricted to their intersections with P. Transitions whose guards become empty in the process, as well as states that become unreachable, are removed. On the other hand it may happen that the reachability graph contains two or more symbolic states (q, P) and (q, P') that correspond to alternative paths to q, and hence the state will be split in the resulting timed automaton. For example state $0'00$ as an initial state can have all clock valuations with $c_x \geq 0$, but when reached again through the path $x^+y^+z^+x^-y^-z^-$, the value of c_x must always exceed 2. Such state splitting will occur very often in systems such as circuits where there are many "diamonds", that is, two competing events e_1 and e_2 that may happen in both $e_1 \prec e_2$ and $e_2 \prec e_1$ orders and converge to the same state q. If these events reset clocks c_1 and c_2, respectively, the reachability graph will contain two symbolic states, $(q, c_1 \leq c_2)$ and $(q, c_2 \leq c_1)$.

It is not hard to see that the new timed automaton $\mathcal{A}'$ admits exactly the same set of behaviors as the original automaton $\mathcal{A}$, together with an additional evident property that any configurations that satisfies the staying condition of a state is indeed reachable. Every finite or infinite path (a qualitative behavior) in the transition graph of $\mathcal{A}'$ is an untimed abstraction of a *feasible behavior* of $\mathcal{A}$, a behavior that satisfies the timing constraints. If our goal is to verify some untimed property of the system, we can remove the clocks from $\mathcal{A}'$ (after having used them to eliminate infeasible paths) and apply standard untimed verification algorithms. However if we want to compose the system with other components it might not be a good idea to get rid of *all* timing information. The untimed abstraction does not constrain in any way the time between x^+, y^+ and z^+, which can be arbitrarily small or large, and will make it difficult (if not impossible) to prove the correctness of the interaction of the circuit with its environment. An abstraction which maintains *some* of the timing constraints but which has less states and clocks, would be very useful in this context.

4.3 Abstraction by Clock Projection

We want the abstract model to approximate the *timed input-output relationship* maintained by the system. Clock c_x measures the time since the last change in the external input x while clocks c_y and c_z measure time elapsed since the occurrence of *internal events*, the excitation of the two gates, events that are of no interest to the general public. We can thus "hide" these clocks and project the guards and staying conditions on clock c_x to obtain the automaton $\mathcal{A}''$ of Figure 6. Note that a projection of a polyhedron P into a lower dimensional polyhedron P' makes some of the constraints which are implicit (redundant) in P, explicit in P'. For example the polyhedron defined by $1 \leq c_z \leq 2 \wedge 1 \leq c_x - c_z \leq 2$ is projected onto $2 \leq c_x \leq 4$.

When y is not observable outside the system, the set of all xz behaviors of $\mathcal{A}''$ is exactly that of $\mathcal{A}$ and $\mathcal{A}'$ and no information is lost. Unfortunately, in the general case, the projection of clocks does lose information. Consider the same circuit but with y visible to the external world. In this case $\mathcal{A}''$ is an over approximation because it allows a behavior like $5 \cdot x^+ \cdot 1 \cdot y^+ \cdot 3 \cdot z^+$, where y "chooses" to change in the earliest time $t \in [1, 2]$ after x while z is allowed to chooses the largest element in $[2, 4] = [1 + 1, 2 + 2]$ while in $\mathcal{A}$ and $\mathcal{A}'$ its choices were restricted to the interval $[t + 1, t + 2]$. This is the type of accuracy we are ready to sacrifice for the purpose of complexity reduction.

The outcome of our abstraction technique is a timed automaton over the inputs and outputs of the system, where output transition guards involve clocks that measure the time elapsed since the occurrence of input events. In the previous example we used input clock c_x which was reset at *every* change in x. This construction was correct because the *variability constraint* prevented the arrival of an x-event while the circuit is still busy "digesting" the previous event. When this constraint is relaxed, an x-labeled transition may be taken in a state where one or more gates excited by the previous x-transition have not yet stabilized. In our example, if we change the variability constraint from $c_x \geq 5$ to $c_x \geq 3$, x may change at state $1'10'$ where y has already stabilized but z is still excited by the previous change. If we reset c_x we lose the time of that previous event, and when we project transitions guards on c_x we *do not* express the temporal distance between the rising of z and its *triggering event*.[5]

To guarantee correct abstraction each input event should reset *its proper clock* which will stay active as long as the "wave" of reactions it triggered has not propagated through the system. Within our modeling methodology, the number of input events that may be active simultaneously in an acyclic system is bounded and hence a finite number of clocks will suffice to retain the information necessary for relating the timing of input and output events. To implement these input clocks we modify the timed automaton model to include a pool of *dynamic clocks* which are activated by input events and killed when the effect of these events propagates to the output. The attachment of these clocks to input events is not fixed and the same clock can, for example, denote at some point the time elapsed since the oldest x_1 event still in the system, and at some other point, the time since the most recent x_2 event. Technically speaking, we replace the input generator by one which creates a new clock at every transition, and keeps track of the input events that are still alive in the system and the clocks that represent them. It is worth mentioning that such dynamic clocks are useful in other, more theoretical, contexts [8].

4.4 Minimization

By hiding internal clocks we obtain an abstract model whose number of clocks need not be equal to the number of timed ele-

[5] In our previous work [3] we have applied this abstraction technique to systems whose inputs changes *only once* at time zero, so that one additional clock was sufficient to project on.

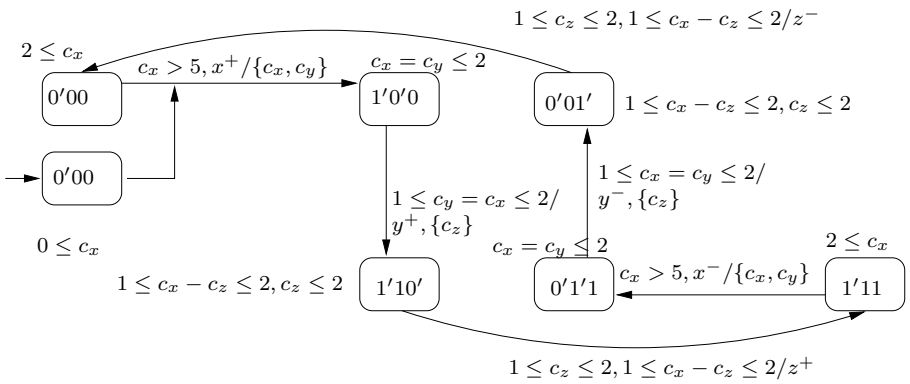


Figure 5: The reachability graph of the automaton in Figure 4, interpreted as a timed automaton $\mathcal{A}'$.

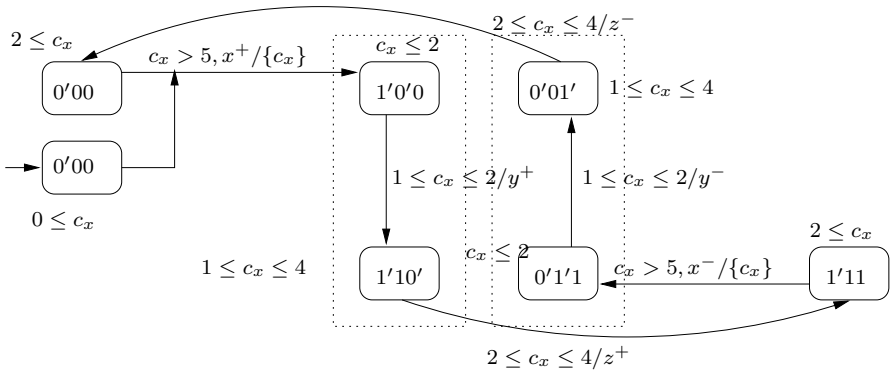


Figure 6: The automaton $\mathcal{A}''$ obtained from $\mathcal{A}'$ by hiding clocks c_y and c_z. The dotted boxes group together states that differ only by the value if the internal variable y.

ments but rather depends on the maximal number of input events that may be "alive" simultaneously in the system. The number of such events depends, of course, on the size of the system as well as on other properties such as the number of inputs, their variability as well as structural properties such as width vs. depth (sequentiality vs. parallelism). Under reasonable assumptions concerning these parameters, the reduction in the number of clocks is significant.

The final step in our procedure aims at reducing the number of *discrete states* by merging states that are equivalent or approximately equivalent in terms of the observable behaviors they admit. Candidates for merging are states that differ from each other only by values of internal variables and of clocks, for example states such as $1'0'0$ and $1'10'$ in automaton $\mathcal{A}''$ of Figure 6, after hiding y. A commonly-used minimization rule (also for untimed systems) is the following.

Let q be a source state for several paths, each consisting of a sequence of unobservable transitions, except the last transition which changes one observable variable and goes to state q'. In this case q and all the intermediate states can be collapsed into one state whose staying condition is the union of those of all states, and which has a transition to q' guarded by the union of all transition guards to q' from the intermediate states. Applying this rule we obtain the automaton of Figure 7 which is nothing but a demonstration of the following equivalence on delay operators: $D_{[1,2]}(D_{[1,2]}(x)) = D_{[2,4]}(x)$. A similar transformation was presented in [11] for timed Petri nets.

The situation gets more complicated when the system admits more parallelism and input events may appear more frequently. We have developed a variety of minimization algorithms that are similar in spirit to those described in [5]. We employ a variety of progressively more "liberal" criteria that merge states which: 1) Admit exactly the same sequences of observable transitions and guards; 2) Differ in guards but the guards are included in each other; 3) Differ in guards and the guard of the new state is the *convex hull* of the guards of the original states; 4) Differ in the order of some sequence of events that admit.

We have implemented all the abovementioned features into a new experimental version of the verification tool IF [4], including an automatic translation from a circuit description language to timed automata, generation and maintenance of dynamic clocks, projection and minimization. The software implementing this technique consists of more than 15000 lines of C++ code.

5. EXPERIMENTAL RESULTS

To assess our approach we applied it first on some classes of synthetic circuits, the first of which is a family of k-long buffers like the one described in the example, with delays in $[3, 5]$. We performed the experiments with two versions of the buffer, one where only the output of gate k is observable, and the other where the output of gate $k/2$ is visible as well. Table 1 shows the results of applying our technique while assuming input variability bounded by 40. Column w shows the maximal number of input events that may be alive in the buffer, which ranges from 1 to 3 depending on the circuit depth. The first pair of columns shows the number of symbolic states and transition in the computed reachability graph. The rest of the table shows the size of the reduced graph using three minimization criteria: *Hidemin* indicates merging only in the case of identical guards, *TimedMin* merges states when guards are included in each other while *Temporal Min* ignores guards and considers equivalence with respect to transition labels.

The other class of examples is inspired by recent research on performance analysis of embedded software, e.g. [10]. We consider systems that generate different types of tasks with some bounded frequency. Each type of task has to go through a partially-ordered set of treatments. Each type of treatment requires a specific resource (machine) for some duration with the possibility of resource conflicts between tasks. These conflicts are resolved by a scheduler applying a simple policy. Each task type has a dedicated bounded buffer. We have applied our technique to an instance of this problem with 2 task types, 3 machines, a priority-based scheduler and parameters that allow 3 events to be alive simultaneously in the system. An unoptimized version of IF generates a reachability graph with 1282 states and 1975 transitions. The version of IF that we use, with dynamic clocks and various optimization that we do not bother to detail, yields a graph with 127 states and 205 transition. After minimization with zone inclusion we obtain the automaton of Figure 8 with 18 states and 33 transitions. Transitions in the reduced model correspond to arrivals of new tasks and their termination.

6. DISCUSSION

We have developed a new promising technique for automatic generation of abstractions for open timed systems. Timed automata with dynamic input clocks may turn out to be the appropriate formalism for characterizing the timed input-output behaviors of complex systems, whose approximation by nice analytical expressions is too coarse. Our technique can also be part of a divide-and-conquer methodology where abstract models of sub-systems are composed together in order to verify a system too large to be analyzed as a whole. Much more experimentation and fine tuning are needed, however, in order to assess the applicability of our approach.

Our original ambitious aim was to provide a "fully-open" abstraction without assuming *any restriction* on the inputs, and letting this restrictions come from each particular environment with which the abstract model is to be composed. However, we have learned in the process that unrestricted inputs generate too many simultaneous waves that lead to explosion. One should be careful, though, not to confuse what is assumed and what should be guaranteed.

7. REFERENCES

[1] R. Alur and D.L. Dill, A Theory of Timed Automata, *Theoretical Computer Science* 126, 183-235, 1994.

[2] E. Asarin, P. Caspi and O. Maler, Timed Regular Expressions, *The Journal of the ACM* 49, 172-206, 2002.

[3] R. Ben Salah, M. Bozga and O. Maler, On Timing Analysis of Combinational Circuits, *FORMATS'03*, 204-219, LNCS 2003.

[4] M. Bozga, S. Graf and L. Mounier, IF-2.0: A Validation Environment for Component-Based Real-Time Systems, *CAV'02*, LNCS 2404, Springer, 2002.

[5] C. Daws and S. Tripakis, Model Checking of Real-Time Reachability Properties using Abstractions, *TACAS'98*, LNCS 1384, 1998.

[6] T. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine, Symbolic Model-checking for Real-time Systems, *Information and Computation* 111, 193-244, 1994.

[7] O. Maler and A. Pnueli, Timing Analysis of Asynchronous Circuits using Timed Automata, *CHARME'95*, 189-205, LNCS 987, Springer, 1995.

[8] O. Maler, D. Nickovic and A. Pnueli, Real Time Temporal Logic: Past, Present, Future, *FORMATS'05*, 2-16, LNCS 3829, Springer, 2005.

[9] J. Sifakis and S. Yovine, Compositional Specification of Timed Systems, *STACS'96*, 347-359, LNCS 1046, Springer,

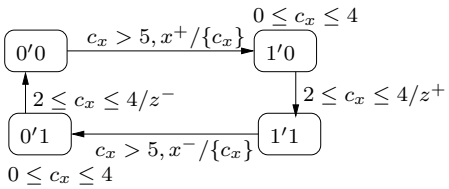


Figure 7: Hiding y and minimizing the automaton

Buff	Time (mn)	w	Generated Graph		HideMin				TimedMin				Temporal Min			
					1 Out		2 Out		1 Out		2 Out		1 Out		2 Out	
			S	T	S	T	S	T	S	T	S	T	S	T	S	T
2	00:00	1	6	6	4	4	6	6	4	4	6	6	4	4	6	6
3	00:00	1	8	8	4	4	6	6	4	4	6	6	4	4	6	6
4	00:00	1	10	10	4	4	6	6	4	4	6	6	4	4	6	6
5	00:00	1	12	12	4	4	6	6	4	4	6	6	4	4	6	6
6	00:00	1	14	14	4	4	6	6	4	4	6	6	4	4	6	6
7	00:00	1	16	16	4	4	6	6	4	4	6	6	4	4	6	6
8	00:00	2	20	22	6	8	8	10	8	10	8	10	8	10	8	10
9	00:00	2	26	32	6	12	8	14	6	8	8	12	6	8	8	12
10	00:00	2	44	62	8	24	10	26	8	12	10	14	6	8	8	10
11	00:01	2	86	132	10	50	12	52	10	18	12	20	6	8	8	12
12	00:03	2	166	266	12	92	18	98	12	26	16	32	6	8	12	18
13	00:20	2	382	624	16	172	26	188	14	36	18	42	6	8	12	18
14	00:34	2	584	966	22	280	44	322	20	84	30	110	6	8	22	48
15	00:54	2	804	1336	26	398	54	446	24	110	42	150	6	8	30	78
16	03:45	3	2208	3846	67	884	125	1109	52	270	96	445	29	85	68	254
17	09:28	3	4349	8284	333	5596	497	4884	235	2590	363	2591	114	881	221	1587
18	38:45	3	12425	25329	1051	39940	1387	28993	623	15375	879	14080	466	14805	756	12974

Table 1: The result of applying our technique to chains of buffers.

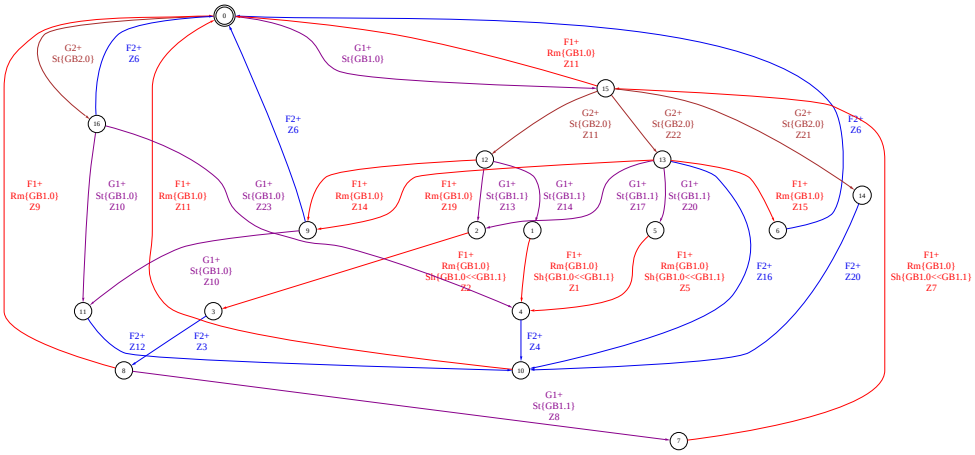


Figure 8: The abstract model of the 2-task, 3-machines problem. Arrival and termination of tasks are denoted by G_i and F_i labels, respectively, whike creation, removal and shifting of clocks by St, Rm and Sh. Zones appear in a separate file to facilitate readability. A more detailed description of the input and the output can be found in www-verimag.imag.fr/∼maler/cav-appendix.html

1996.

[10] E. Wandeler, A. Maxiaguine, L. Thiele: Quantitative Characterization of Event Streams in Analysis of Hard Real-Time Applications, *Real-Time Systems* 29, 205-225, 2005.

[11] H. Zheng, E. Mercer, and C.J. Myers, Modular verification of timed circuits using automatic abstraction, *IEEE Trans. on CAD* 22, 2003.

SAVCBS 2007 CHALLENGE PROBLEM SOLUTIONS

Challenge Problem: Subject-Observer Specification with Component-Interaction Automata

Pavlína Vařeková*, Barbora Zimmerova*
Faculty of Informatics
Masaryk University
602 00 Brno, Czech Republic
{xvareko1, zimmerova}@fi.muni.cz

ABSTRACT

This paper presents our solution to the *Subject-Observer Specification* problem announced as the challenge problem of the *SAVCBS 2007* workshop. The text consists of two parts. In the first part, we present the model of the Subject-Observer system in terms of Component-interaction automata. In the second part, we present our approach to verification of the system model with respect to unlimited number of Observers.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification

General Terms

Component-based systems, verification, specification

Keywords

Component-based systems, dynamic number of components, finite-state systems, verification, software modelling

1. INTRODUCTION

The following solution to the *Subject-Observer Specification* challenge problem is based on the paper [3] that is going to be presented at the workshop. For this reason we do not repeat the definitions given in the paper and reference the reader to the paper. The model is created using the *Component-interaction automata* modelling language (first presented in [1]). For more information on the language please see [4] or the coming detailed case study [5], our result in the *CoCoME (Common Component Modelling Example) Contest*[1], where we have first experienced the Subject-Observer modelling problem.

*The authors have been supported by the grant No. 1ET400300504.

[1] `http://www.cocome.org/`

Sixth International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2007), September 3-4, 2007, Cavtat near Dubrovnik, Croatia.

2. SPECIFICATION

Consider the assignment of the challenge problem. It states that there may be many Observers for a Subject and an Observer may be registered with more than one Subject. But it does not discuss whether the numbers of Subjects and Observers are fixed or change at run-time. While working on the *CoCoME*, we have observed that in practical applications, the number of Subjects is usually fixed, but the number of Observers can grow dynamically. In this example, we suppose the same. In addition, as distinct to the official assignment, we add a possibility of the Observer to deregister from Subjects to make the solution more interesting.

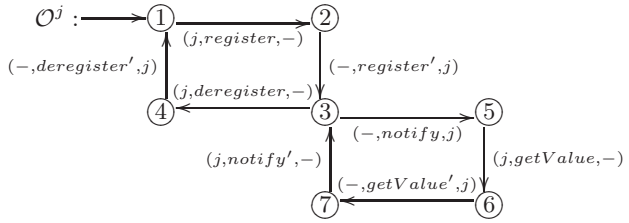
Now we present our model for the example created using Component-interaction automata. For clearness, we start with the model of the system with one Subject only. Then we generalize the model to multiple Subjects.

2.1 Model with one Subject

The model of the system with one Subject and multiple Observers is the following.

2.1.1 Observer

The Observer (in this case with a component name $j \in \mathbb{N}$) first needs to register to get to the state 3 where it can accept notifications and ask for the value managed by the Subject. In the model, each method, e.g. `register()`, is assigned a tuple of action names: `register` denotes the *call* of the method, and `register'` the *return* from the method. These two determine the start and the end of the method's execution.

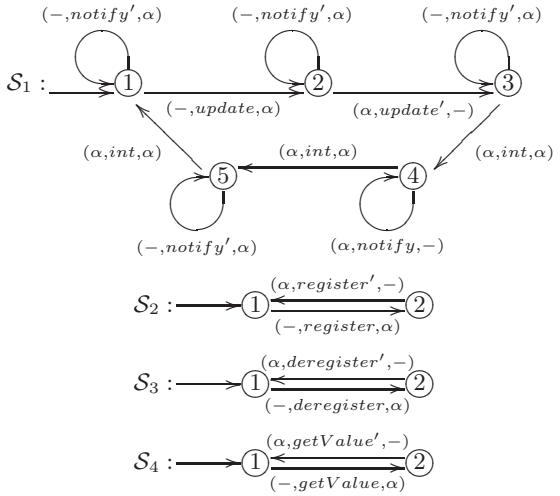


A hierarchy of component names: (j)

Figure 1: A CI model of the Observer $\mathcal{O}^j$

2.1.2 Subject

The Subject $\mathcal{S}$ (component name α) implements four methods, `update()`, `register()`, `deregister()`, `getValue()`, and hence its model consists of four parts connected via the full composition operator $\otimes$ (no transitions removed) as $\mathcal{S} = \otimes\{\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3, \mathcal{S}_4\}$. Models of the parts are in figure 2.



A hierarchy of component names: (α)

Figure 2: A CI model of the Subject $\mathcal{S}$ parts

On the `update()` method (automaton $\mathcal{S}_1$), the Subject first receives the method call, confirms its return (to allow the updater to continue its execution while the notifications are delivered, which is common in Subject-Observer communicational models) and then takes care about notifying the Observers. This proceeds in two loops separated by internal actions. The first loop distributes the notification to the Observers, the second confirms termination of notifications. This allows the Observers to execute bodies of their methods in parallel. More, the confirmation $(-, notify', \alpha)$ is allowed also in other states than 5. This protects the system from deadlock of the Observers that do not manage to synchronize with the Subject before it leaves the state 5. Note that the composition with Observers using a handshake-like composition (required synchronization of complementary labels, which are removed and only synchronized internal labels remain) includes paths representing that 0, 1, 2, ...,all *registered* Observers are notified. However no Observer can be notified twice. This confirms to the *at most once* constraint, which will be verified later in this text.

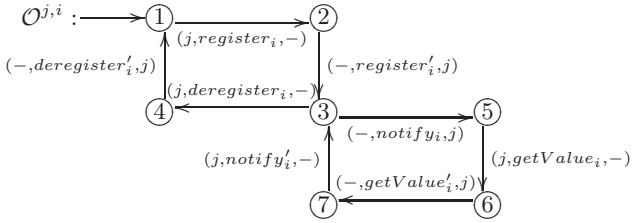
On the remaining methods `register()`, `deregister()`, `getValue()` (automata $\mathcal{S}_2, \mathcal{S}_3, \mathcal{S}_4$), the Subject only receives the call and then returns. More interesting behaviour could be inserted on the place of state 2.

2.2 Model with several Subjects

The presented model can be readily extended to the multi-Subject case. Suppose the number of Subjects is n, then we have Subjects $\mathcal{S}^i$ for $i \in \{1, 2, ..., n\}$ where i represents the *id* of the Subjects. We add this *id* also to the names of methods to distinguish which Subject an Observer wants to communicate with.

2.2.1 Observer

The Observer now consists of n parts identical up to indexes in actions, each one for communication with one Subject. A model of one part is in figure 3. The parts are again connected via the full composition operator $\otimes$, hence the model of the Observer (component name j) is $\mathcal{O}^{j,\{1,...,n\}} = \otimes\{\mathcal{O}^{j,i}\}_{i\in\{1,...,n\}}$.

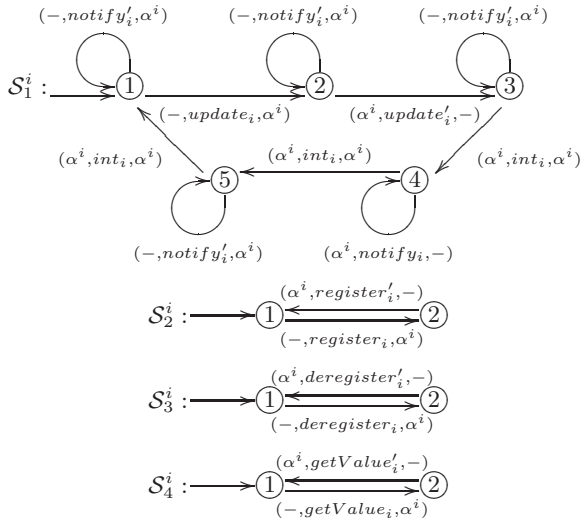


A hierarchy of component names: (j)

Figure 3: A CI model of the Observer part $\mathcal{O}^{j,i}$

2.2.2 Subject

The model of a Subject $\mathcal{S}^i$ (component name α^i where $\alpha^i, i \in \mathbb{N}$, denotes the sequence of i symbols α) is analogical to the model of Subject $\mathcal{S}$ (figure 2). It again consists of four parts $\mathcal{S}^i = \otimes\{\mathcal{S}^i_1, \mathcal{S}^i_2, \mathcal{S}^i_3, \mathcal{S}^i_4\}$. The models of the parts are in figure 4.



A hierarchy of component names: (α^i)

Figure 4: A CI model of the Subject $\mathcal{S}^i$ parts

2.2.3 The composite model

Now we can fix the number of Subjects to n, and the number of Observers to m, hence get automata $\mathcal{S}^1, \mathcal{S}^2, \ldots, \mathcal{S}^n$ and $\mathcal{O}^{1,\{1,2,...,n\}}, \mathcal{O}^{2,\{1,2,...,n\}}, \ldots, \mathcal{O}^{m,\{1,2,...,n\}}$ given by description above, and compose them together using handshake-like composition. The handshake-like composition requires synchronization of those labels that have a counterpart at other component. Such labels are afterwards removed from the composition and only the internally synchronized labels are left.

This type of composition can be realized via composition operator $\otimes^{\mathcal{F}}$ (see [3]) where $\mathcal{F} =$
$\bigcup_{i,j\in\mathbb{N}}\{_{(\alpha^i,notify_i,j),(j,notify'_i,\alpha^i),(j,register_i,\alpha^i),(\alpha^i,register'_i,j),}$
$_{(j,deregister_i,\alpha^i),(\alpha^i,deregister_i,j),(j,getValue_i,\alpha^i),(\alpha^i,getValue'_i,j)}\}$
$\cup \bigcup_{i\in\mathbb{N}}\{_{(-,update_i,\alpha^i),(\alpha^i,update'_i,-),(\alpha^i,int_i,\alpha^i)}\}$
The resulting model is then:
$\otimes^{\mathcal{F}}\{\mathcal{S}^1,\ldots,\mathcal{S}^n,\mathcal{O}^{1,\{1,2,\ldots,n\}},\ldots,\mathcal{O}^{m,\{1,2,\ldots,n\}}\}$.

Such a model is suitable for verification of properties fixed to the selected numbers of Subjects and Observers. However we are interested also in verification of the properties for an arbitrary number of Observers, because it is usual that the Observers can be added to and removed from the system dynamically. To this issue, we may apply the solution [3] presented at this workshop. However before we can do so, we need to adjust the system to the structure that is awaited by our approach. That is that the system consists of one stable component (called *provider*) and a number of dynamic components of one type (called *clients*). To meet this constraint, we only compose all Subjects into a composite Subject managing all values $\mathcal{S}^{\{1,\ldots,n\}} = \otimes\{\mathcal{S}^i\}_{i\in\{1,\ldots,n\}}$ and define the dynamic system model as $\mathcal{S}^n\&\mathcal{O} = (\mathcal{S}^{\{1,\ldots,n\}},\{\mathcal{O}^{j,\{1,\ldots,n\}}\}_{j\in\mathbb{N}},\mathcal{F})$ and a composite system with m Observers as $\mathcal{S}^n\&\mathcal{O}_m = \otimes^{\mathcal{F}}\{\mathcal{S}^{\{1,\ldots,n\}},\mathcal{O}^{1,\{1,\ldots,n\}},\ldots,\mathcal{O}^{m,\{1,\ldots,n\}}\}$.

Now the composite Subject represents the *provider* and Observers the *clients* of the dynamic system we have just defined.

3. VERIFICATION

In this section, we describe the verification of a dynamic system $\mathcal{S}^n\&\mathcal{O}$ for arbitrary fixed number of Subjects n. We present the application of the approach introduced in [3] for effective verification of properties expressed as sequences $Property(\mathcal{S}^n\&\mathcal{O},m)$. Then we illustrate the technique on the properties of the systems $\mathcal{S}^1\&\mathcal{O}$ and $\mathcal{S}^2\&\mathcal{O}$.

3.1 Properties for verification

In formal verification techniques, like *model checking* [2], the properties for verification are specified in temporal logics. In our approach, we use the logic CI-LTL. CI-LTL is an extension of the action-based LTL, which is in addition to expressing that an action (label) l is proceeding $\mathcal{P}(l)$ able to express that a given label l is enabled $\mathcal{E}(l)$ in a state of a path (one-step branching). See [3] for description of CI-LTL.

Assume a dynamic system $\mathcal{D}$. The properties that we aim to verify, can be specified with a sequence of formulas $\{\varphi_i\}_{i\in\mathbb{N}_0}$ over $\mathcal{L}_{\mathcal{D}}$ such that a property is satisfied iff for each $i\in\mathbb{N}_0$ it holds that $\varphi_i \models \mathcal{D}_i$. Note that not every sequence of formulas $\{\varphi_i\}_{i\in\mathbb{N}_0}$ represents a meaningful property of the system. Thus we concentrate on the formulas satisfying the following.

(1) The property makes no distinctions among clients.
(2) If the property is violated by a path in a system $\mathcal{D}_{i+j}$ where j components do not perform any steps, the property is violated by the same sequence of labels also in the system with i clients only.

Moreover, we pose the following two restrictions. See [3] for proper description of these.

(3) We focus only on the formula sequences that represent properties whose violation involves a finite number m of observed components only.
(4) We consider only the properties that are invariant under stuttering according to CI-LTL.

The set of properties that fulfills these conditions for model $\mathcal{D}$ and the finite number of components m is denoted $Property(\mathcal{D},m)$.

3.2 Verification technique

The core idea of the verification process is based on finding a number $k\in\mathbb{N}$ such that if the property is violated on a system then it must be violated for the system with maximally k clients deployed. If such k exists, it allows us to reduce verification of the infinite system to a finite one. In particular, to a verification of a finite number of finite systems – with $0,1,\ldots,k$ clients deployed. The value k for a dynamic system $\mathcal{D}$ and a set of observable actions X can be estimated as a sum of two measures. The first one is a measure of complexity of a dynamic system reflecting the maximal number of clients that are regarded by the provider. The second one is a similar measure on properties that reflects the minimal number of clients necessary to exhibit a path violating studied property.

3.3 Optimizations

The model of the dynamic system for the Subject-Observer problem $\mathcal{S}^n\&\mathcal{O}$ does not exactly follow the pattern client-provider supposed by our approach [3]. In [3] we require that the provider of a dynamic system can in any time regard at most n clients for a constant $n\in\mathbb{N}_0$ [2]. In the case of a dynamic system $\mathcal{S}^n\&\mathcal{O}$, the client (Observer) is regarded (served) if and only if it is registered. Note that the number of registered Observers in the system $\mathcal{S}^n\&\mathcal{O}$ can be arbitrary. Hence $|\mathcal{S}^n\&\mathcal{O}|_{\mathcal{L}_{\mathcal{C}_0}} = \infty$. Therefore we need to use two optimizations that lead to $|\mathcal{S}^n\&\mathcal{O}|_X$ finite.

3.3.1 Optimization 1.

With regard to the presence of sub-formulas $\mathcal{P}(l)$ and $\mathcal{E}(l)$ in the verified sequence of formulas, we minimize the set X used in the computation of $|\mathcal{D}|_X$. This decrease of the number of observed labels increases the probability that w.r.t. these labels the provider (combined Subject) regards at most a fixed number of clients (Observers) at any time, which makes $|\mathcal{D}|_X$ finite.

Example 3.1. Suppose a dynamic system $\mathcal{S}^1\&\mathcal{O}$ and two sets of observable labels X for which we compute the value $|\mathcal{S}^1\&\mathcal{O}|_X$:

• $X = \{(\alpha,notify_1,-),(-,notify'_1,\alpha)\}$
In this case, the Observer is in a cycle of service if and only if it started and did not finish the notification. The number

[2]It could be said that a client is regarded (or served) by the provider if it synchronized with the provider on an observable label that started client's execution (cycle of service) and has not yet synchronized on an observable label representing the end of this execution. It holds that the value $|\mathcal{D}|_X$ is always grater than the maximal number of served clients.

of such Observers is not bound. It holds that the maximal number of served Observers is lower or equal to $|\mathcal{S}^1\&\mathcal{O}|_X$, hence $|\mathcal{S}^1\&\mathcal{O}|_X = \infty$.

• $X = \{(-, notify'_1, \alpha)\}$
In this case no Observer can be in a cycle of service. Roughly speaking this is because X implies only one observable transition and hence no Observer can get in between of two observable transitions that bound a cycle of service (for definition of a cycle of service see [3]). Therefore it is possible that the value $|\mathcal{S}^1\&\mathcal{O}|_X$ is finite. After computing this value we get $|\mathcal{S}^1\&\mathcal{O}|_X = 1$.

3.3.2 Optimization 2.

This optimization is based on a modification of the dynamic system and the property in a way that thanks to them we may verify the original property on the original system, and we increase the probability that the measure of complexity of the modified model with respect to the modified set of properties and the set of observable labels is finite. Let $\{\varphi_i\}_{i\in\mathbb{N}_0} \in Property(\mathcal{D}, m)$. Suppose:
• A dynamic system $\overline{\mathcal{D}}$ created from system $\mathcal{D}$ by modification of its provider – modelling the provider of the system $\mathcal{D}$ composed with m clients (see figure 5). The remaining items of the dynamic system are identical to the system $\mathcal{D}$.
• A sequence of properties $\{\overline{\varphi}_i\}_{i\in\mathbb{N}_0}$ such that the formula $\overline{\varphi}_n$ captures for the automaton $\overline{\mathcal{D}}_n$ the same property as the formula φ_{n+m} narrowed down to the clients $1, ..., m$ for the automaton $\mathcal{D}_{n+m}$.

Example 3.2. The formulas $\{\varphi_i\}_{i\in\mathbb{N}_0} \in Property(\mathcal{D}, 1)$ for instance capture the property:
After every start of a notification (sent to any Observer) there follows the finish of this notification.
Then the formulas $\{\overline{\varphi}_i\}_{i\in\mathbb{N}_0} \in Property(\overline{\mathcal{D}}, 0)$ capture the property:
After every start of a notification sent to an Observer that is modelled as a part of the provider, there follows the finish of this notification.

As $\{\varphi_i\}_{i\in\mathbb{N}_0} \in Property(\mathcal{D}, m)$ and these formulas make no distinction among clients, it holds that:

$$\overline{\mathcal{D}}_n \models \overline{\varphi}_n \text{ iff } \mathcal{D}_{n+m} \models \varphi_{n+m}.$$

As $\{\overline{\varphi}_i\}_{i\in\mathbb{N}_0} \in Property(\overline{\mathcal{D}}, 0)$ then if $\overline{\mathcal{D}}_0 \models \overline{\varphi}_0, \ldots, \overline{\mathcal{D}}_{|\overline{\mathcal{D}}|_{X'}} \models \overline{\varphi}_{|\overline{\mathcal{D}}|_{X'}}$ (for appropriate X') it follows from the intuition presented in Optimization 2 that it must hold that $\overline{\mathcal{D}}_n \models \overline{\varphi}_n$ for every $n \in \mathbb{N}_0$. From lemmas in [3] we get that for every $n \in \mathbb{N}_0$ it holds that $\mathcal{D}_{n+m} \models \varphi_{n+m}$.
Moreover, the set X' contains less external labels (input and output labels) than the set X, and hence there is higher probability that $|\overline{\mathcal{D}}|_{X'}$ is finite.

Example 3.3. Suppose that for the system $\mathcal{S}^1\&\mathcal{O}$ we are interested in the verification of the properties from example 3.2. These properties can be captured as formulas $Property(\mathcal{S}^1\&\mathcal{O}, m)$ and for their verification, it suffices to use the set of observable labels $X = \{(\alpha, notify_1, -), (-, notify'_1, \alpha)\}$. From the reasoning above, we know that $|\mathcal{S}^1\&\mathcal{O}|_X = \infty$. However if we consider the model $\overline{\mathcal{S}^1\&\mathcal{O}}$ where the provider contains not only n components for the Subjects, $n = 1$ here, but also m components representing Observers, $m = 1$ here (modelled with component name β), the minimal set of observable actions is $X' = \{(\alpha, notify_1, \beta), (\beta, notify'_1, \alpha)\}$ and it holds that $\overline{|\mathcal{S}^1\&\mathcal{O}|}_{X'} \leq 1$.

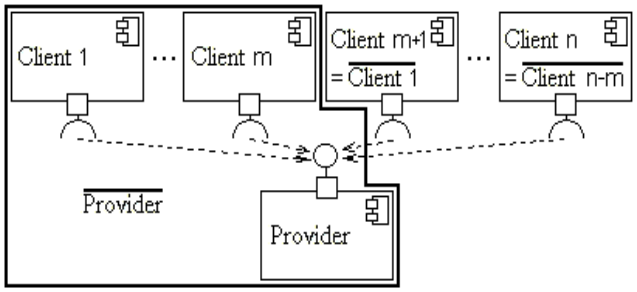


Figure 5: A dynamic system $\mathcal{D}$ with n clients and dynamic system $\overline{\mathcal{D}}$ with $n - m$ clients

3.4 Algorithm for verification

The next section presents examples of the properties of the Subject-Observer system that are interesting for verification. Validity of these properties can be showed by intuitive reasoning. However for more complex properties, the automatic verification technique is necessary. In this section we present such a technique based on the results from [3]. This verification will take advantage of the optimizations introduced above. It consists of the following three steps.

(1) Creation of a harmonized sequence of formulas $\{\varphi_i\}_{i\in\mathbb{N}_0}$ that corresponds to a given property for the dynamic system $\mathcal{S}^n\&\mathcal{O}$. Detecting whether there is a finite m such that the sequence of formulas $\{\varphi_i\}_{i\in\mathbb{N}_0}$ is part of the set $Property(\mathcal{S}^n\&\mathcal{O}, m)$. If there is no such m, it is not possible to verify $\{\varphi_i\}_{i\in\mathbb{N}_0}$ using the approach from [3]. It is clear that a property can be captured with various formulas, some of them can be verified using our approach, some of them cannot (see the property (c) in section 4.1).

(2) Modification of the dynamic system $\mathcal{S}^n\&\mathcal{O}$ and the sequence of formulas $\{\varphi_i\}_{i\in\mathbb{N}_0}$ using Optimization 2 to the dynamic system $\overline{\mathcal{S}^n\&\mathcal{O}}$ (the Observers that become part of the provider will be referred to as $\beta, \beta\beta, \ldots$) and the sequence of formulas $\{\overline{\varphi}_i\}_{i\in\mathbb{N}_0}$. With respect to the Optimization 1, we select the set X of observable labels and we compute $|\overline{\mathcal{S}^n\&\mathcal{O}}|_X$.

(3) The dynamic system $\overline{\mathcal{S}^n\&\mathcal{O}}$ and formulas $\{\overline{\varphi}_i\}_{i\in\mathbb{N}_0}$ agree with the prerequisites of Lemma 5.4 from [3]. Hence for the verification of the given property, it suffices to check the validity of the according formulas on the systems $\overline{\mathcal{S}^n\&\mathcal{O}}_0$, $\overline{\mathcal{S}^n\&\mathcal{O}}_1$, ..., $\overline{\mathcal{S}^n\&\mathcal{O}}_{|\overline{\mathcal{S}^n\&\mathcal{O}}|_X}$. If the verification succeeds, it is verified that the property holds for all systems $\overline{\mathcal{S}^n\&\mathcal{O}}_0$, $\overline{\mathcal{S}^n\&\mathcal{O}}_1$, ... Therefore from the discussion in Optimization 2 it follows that the original property is verified on models $\mathcal{S}^n\&\mathcal{O}_m$, $\mathcal{S}^n\&\mathcal{O}_{m+1}$, ... For verification of the whole dynamic system, we more need to verify the original property on $\mathcal{S}^n\&\mathcal{O}_0$, $\mathcal{S}^n\&\mathcal{O}_1$, ..., $\mathcal{S}^n\&\mathcal{O}_{m-1}$.

Only the first step (1) needs to be supported manually, the steps (2) and (3) can be done automatically.

The intuition for getting the over-approximation of the value $|\mathcal{D}|_X$, which we will use together with Lemma 5.4 from [3] for automatization of the steps (2) and (3) from the algorithm above is based on a simple observation:

"If for a dynamic system $\mathcal{D}$ the automaton $\mathcal{D}_n$ generates all possible runs with respect to the observable labels X, then for every $i \in \mathbb{N}$ the automaton $\mathcal{D}_{n+i}$ generates again the same runs. Hence it holds that $|\mathcal{D}|_X \leq n$."

4. EXAMPLES

Now we present several examples to demonstrate the verification of the Subject-Observer system model.

4.1 Verification of the syst. with one Subject

In this section, we illustrate the verification technique presented above on the dynamic system $\overline{\mathcal{S}^1\&\mathcal{O}}$. In the examples, we discuss the first step in detail because it is the manual part of the verification. For the remaining two steps, which can be done automatically, we just present the results without further discussion.

a) If a run contains infinitely many steps concerning some Observer, then the Observer is infinitely many times enabled to receive notifications.

- This property can be expressed by a harmonized set of formulas $\{\varphi_i\}_{i\in\mathbb{N}_0}$:
  $\varphi_i = \bigwedge_{j\leq i} \varphi(\alpha, j)$,
  where
  $\varphi(\alpha, j) = [\mathcal{G}\,\mathcal{F}\,\bigvee_{l\in Lab_j} \mathcal{P}(l)] \Rightarrow [\mathcal{G}\,\mathcal{F}\,\mathcal{E}(\alpha, notify_1, j)]$,
  $Lab_j = \{_{(j,register_1,\alpha),(\alpha,register'_1,j),(j,deregister_1,\alpha),}$
  $_{(\alpha,deregister'_1,j),(\alpha,notify_1,j),(j,notify'_1,\alpha),(j,getValue_1,\alpha),}$
  $_{(\alpha,getValue'_1,j)}\}$
  (Lab_j is a set of all the labels that are present in some of the automata $\mathcal{S}^1\&\mathcal{O}_j$, $\mathcal{S}^1\&\mathcal{O}_{j+1}$, ... concerning the Observer $\mathcal{O}^j$).
  For any $i \in \mathbb{N}_0$ and an infinite run π starting in an initial state of the automaton $\mathcal{S}^1\&\mathcal{O}$ satisfying $\pi \not\models \varphi_i$, there exists a number $j \in \mathbb{N}$ such that on this run the formula $\varphi(\alpha, j)$ is not valid. For confirming this violation it suffices to observe the Subject and the Observer with the numerical name j. Hence in general, it always suffices to observe one Observer to confirm the violation of the property. It holds that:
  $\{\varphi_i\}_{i\in\mathbb{N}_0} \in Property(\mathcal{S}^1\&\mathcal{O}, 1)$

- The modified sequence of formulas $\{\overline{\varphi}_i\}_{i\in\mathbb{N}_0}$ is then
  $\overline{\varphi}_i = \varphi(\alpha, \beta)\ \ \forall i \in \mathbb{N}_0$.
  In this case, it suffices to compute the value $|\overline{\mathcal{S}^1\&\mathcal{O}}_{Lab_\beta}|$ (see Lemma 5.4 from [3])
  and from the algorithm discussed above it follows that:
  $|\overline{\mathcal{S}^1\&\mathcal{O}}_{Lab_\beta}| \leq 0$.

- Then after verifying the models $\mathcal{S}^1\&\mathcal{O}_0$ and $\overline{\mathcal{S}^1\&\mathcal{O}_0}$ we can conclude that the property always holds.

b) If one of the registered Observers receives a notification and some other Observer is also ready to receive one (is registered and has not receive it yet), it will receive the notification too.

- This property can be expressed by a harmonized set of formulas $\{\varphi_i\}_{i\in\mathbb{N}_0}$:
  $\varphi_i = \bigwedge_{j_1,j_2\leq i, j_1\neq j_2} \varphi(\alpha, j_1, j_2)$,
  where
  $\varphi(\alpha, j_1, j_2) = \mathcal{G}\ [(\mathcal{P}(\alpha, notify_1, j_1) \wedge \mathcal{E}(\alpha, notify_1, j_2)) \Rightarrow (true\ \mathcal{U}\ \mathcal{P}(\alpha, notify_1, j_2))]$
  For any $i \in \mathbb{N}_0$ and an infinite run π starting in an initial state of the automaton $\mathcal{S}^1\&\mathcal{O}$ satisfying $\pi \not\models \varphi_i$, there exists distinct numbers $j_1, j_2 \in \mathbb{N}$ such that on this run the formula $\varphi(\alpha, j_1, j_2)$ is not valid. For confirming this violation it suffices to observe the Subject and the Observers with the numerical names j_1 and j_2. Hence in general, it always suffices to observe two Observer to confirm the violation of the property. It holds that:
  $\{\varphi_i\}_{i\in\mathbb{N}_0} \in Property(\mathcal{S}^1\&\mathcal{O}, 2)$.

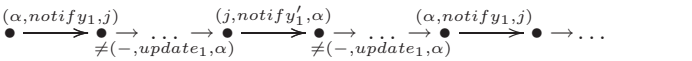
- $\{\overline{\varphi}_i\}_{i\in\mathbb{N}_0}$ satisfies $\forall i \in \mathbb{N}_0\ \overline{\varphi}_i = \varphi(\alpha, \beta, \beta\beta)$.
  In this case, it suffices to compute the value $|\overline{\mathcal{S}^1\&\mathcal{O}}_X|$ for $X = \{_{(\alpha,notify_1,\beta^k),(\beta^k,notify'_1,\alpha),(\alpha,register'_1,\beta^k),}$
  $_{(\beta^k,deregister_1,\alpha)\ \mid\ k\in\{1,2\}}\}$
  (see Lemma 5.4 from [3]) and from the algorithm discussed above it follows that: $|\overline{\mathcal{S}^1\&\mathcal{O}}|_X \leq 0$.

- Then after verifying the models $\mathcal{S}^1\&\mathcal{O}_0$, $\mathcal{S}^1\&\mathcal{O}_1$ and $\overline{\mathcal{S}^1\&\mathcal{O}_0}$ we can conclude on the validity of the formula. In this case the formula does not hold which is confirmed by the model $\overline{\mathcal{S}^1\&\mathcal{O}_0}$.
  The counterexample is the run of the automaton $\overline{\mathcal{S}^1\&\mathcal{O}_0}$ where first the Observers β and $\beta\beta$ register for receiving notifications, then the Subject is updated, it sends the notification to Observer β, but never deliverers the notification to Observer $\beta\beta$ because this gets locked in a loop of registering and deregistering.

c) After any update, each Observer receives at most one notification about value change. This reflects that *"each Observer is called at most once per state change"* **from the assignment of the problem.**

- This property can be expressed by a set of formulas $\{\varphi_i\}_{i\in\mathbb{N}_0}$:
  $\varphi_i = \bigwedge_{j\leq i} \neg\mathcal{F}\ \varphi(\alpha, j)$,
  where
  $\varphi(\alpha, j) = \mathcal{P}(\alpha, notify_1, j) \wedge \mathcal{X}\ [\neg\mathcal{P}(-, update_1, \alpha)\ \mathcal{U}\ \mathcal{P}(\alpha, notify_1, j)]$
  These formulas contain operator $\mathcal{X}$, therefore they are not invariant under stuttering and that is why $\{\varphi_i\}_{i\in\mathbb{N}_0} \notin Property(\mathcal{S}^1\&\mathcal{O}, m)$ holds for any $m \in \mathbb{N}_0$.
  We take advantage of the fact that each Observer after getting the notification must first confirm the end of the notification before it is able to receive another one. Hence we can express the property with the following harmonized set of formulas:
  $\varphi_i = \bigwedge_{j\leq i} \neg\mathcal{F}\ \varphi(\alpha, j)$,
  where
  $\varphi(\alpha, j) = \mathcal{P}(\alpha, notify_1, j) \wedge [\neg\mathcal{P}(-, update_1, \alpha)\ \mathcal{U}\ \{\mathcal{P}(j, notify'_1, \alpha) \wedge [\neg\mathcal{P}(-, update_1, \alpha)\ \mathcal{U}\ \mathcal{P}(\alpha, notify_1, j)]\}]$.

  Example of a run violating the formula $\varphi(\alpha, j)$ is for instance (without names of states):



Using the reasoning analogical to the one above, we can conclude that $\{\varphi_i\}_{i\in\mathbb{N}_0} \in Property(\mathcal{S}^1\&\mathcal{O}, 1)$

- $\overline{\varphi}_i = \neg\mathcal{F}\ \varphi(\alpha,\beta)$.
  We need to compute the value $|\overline{\mathcal{S}^1\&\mathcal{O}}|_{\{(-,update_1,\alpha),(\alpha,notify_1,\beta),(\beta,notify'_1,\alpha)\}}$.
  $|\overline{\mathcal{S}^1\&\mathcal{O}}|_{\{(-,update_1,\alpha),(\alpha,notify_1,\beta),(\beta,notify'_1,\alpha)\}} \leq 1$.

- Then after verifying the models $\mathcal{S}^1\&\mathcal{O}_0$, $\overline{\mathcal{S}^1\&\mathcal{O}}_0$ and $\overline{\mathcal{S}^1\&\mathcal{O}}_1$ we can conclude that the property always holds.

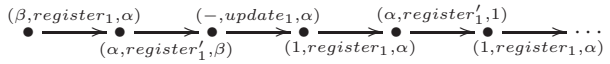
d) Anytime, the Subject is in future enabled to execute `update()`.

- This property can be expressed by a harmonized set of formulas $\{\varphi_i\}_{i\in\mathbb{N}_0}$:
  $\varphi_i = \mathcal{G}\,\mathcal{F}\,\mathcal{E}(-, update_1, \alpha)$.
  As the formulas do not contain a name of any Observer, it holds that $\{\varphi_i\}_{i\in\mathbb{N}_0} \in Property(\mathcal{S}^1\&\mathcal{O}, 0)$.

- $\overline{\varphi}_i = \varphi_i = \mathcal{G}\,\mathcal{F}\,\mathcal{E}(-, update_1, \alpha)$.
  We need to compute the value $|\overline{\mathcal{S}^1\&\mathcal{O}}|_{\{(-,update_1,\alpha),(\alpha,int_1,\alpha)\}}$.
  It holds that $|\overline{\mathcal{S}^1\&\mathcal{O}}|_{\{(-,update_1,\alpha),(\alpha,int_1,\alpha)\}} \leq 1$.

- The verification of the models $\overline{\mathcal{S}^1\&\mathcal{O}}_0$, $\overline{\mathcal{S}^1\&\mathcal{O}}_1$ shows that the property holds on the model $\overline{\mathcal{S}^1\&\mathcal{O}}_0$ but does not hold on the model $\overline{\mathcal{S}^1\&\mathcal{O}}_1$.

  The counterexample is the run of the automaton $\overline{\mathcal{S}^1\&\mathcal{O}}_1$ (for space reasons we do not include the names of the states):



e) If it holds for a run that every update of the Subject is followed by starting the notification of all the Observers, then each update will be also followed by finishing of the notifications by all the Observers.

- This property is an example of a property that can be expressed as a sequence of formulas, but there exists no $m \in \mathbb{N}_0$ such that the sequence of formulas is in $Property(\mathcal{S}^1\&\mathcal{O}, m)$. This follows from the fact that if a property is violated by some run, we are not able to bound the number of clients that always suffices to confirm the faultiness of the run.

4.2 Verification of the syst. with two Subjects

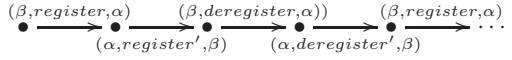
In this section we present examples of two properties discussed above on the dynamic system Subject-Observer with two Subjects $\mathcal{S}^2\&\mathcal{O}$.

f) If a run contains infinitely many steps concerning some Observer, then the Observer is infinitely many times enabled to receive notifications from both Subjects.

- This property can be expressed by a harmonized set of formulas $\{\varphi_i\}_{i\in\mathbb{N}_0}$:
  $\varphi_i = \bigwedge_{j\leq i} \varphi(\alpha, \alpha\alpha, j)$,
  where
  $\varphi(\alpha,\alpha\alpha,j) = [\mathcal{G}\,\mathcal{F}\ \bigvee_{l\in Lab_j} \mathcal{P}(l)] \Rightarrow$
  $[\mathcal{G}\,(\mathcal{F}\,\mathcal{E}(\alpha, notify_1, j)) \wedge (\mathcal{F}\,\mathcal{E}(\alpha\alpha, notify_2, j))]$,
  and
  $Lab_j = \{_{(j,register_k,\alpha^k),(\alpha^k,register'_k,j),(j,deregister_k,\alpha^k),}$
  $_{(\alpha^k,deregister'_k,j),(\alpha^k,notify_k,j),(j,notify'_k,\alpha^k),}$
  $_{(j,getValue_k,\alpha^k),(\alpha^k,getValue'_k,j)}\ |_{k\in\{1,2\}}\}$
  (Lab_j is a set of all the labels that are present in some of the automata $\mathcal{S}^2\&\mathcal{O}_0$, $\mathcal{S}^2\&\mathcal{O}_1$, ... concerning the Observer $\mathcal{O}^j$).
  From the reasons analogical to the ones discussed above, it follows that:
  $\{\varphi_i\}_{i\in\mathbb{N}_0} \in Property(\mathcal{S}^2\&\mathcal{O}, 1)$

- The modified sequence of formulas $\{\overline{\varphi}_i\}_{i\in\mathbb{N}_0}$ is then $\overline{\varphi}_i = \varphi(\alpha, \alpha\alpha, \beta)\ \ \forall i \in \mathbb{N}_0$.
  Hence it suffices to compute the value $|\overline{\mathcal{S}^2\&\mathcal{O}}_{Lab_\beta}|$ and it follows that: $|\overline{\mathcal{S}^2\&\mathcal{O}}_{Lab_\beta}| \leq 0$.

- From the verification of the models $\overline{\mathcal{S}^2\&\mathcal{O}}_0$ we learn that the property is not satisfied. The counterexample is the run containing an infinite number of the transitions concerning the Observer β who can never accept the notification about the update of the value managed by the Subject $\alpha\alpha$ (for space reasons we again do not include the names of the states):



g) After any update, each Observer receives at most one notification about value change. This reflects that *"each Observer is called at most once per state change"* **from the assignment of the problem.**

- Analogically to the property (c) in section 4.1, we take advantage of the fact that each Observer after getting the notification must first confirm the end of the notification before it is able to receive another one. Hence we can express the property with the following harmonized set of formulas (without the operator $\mathcal{X}$):
  $\varphi_i = [\bigwedge_{j\leq i} \neg\mathcal{F}\ \varphi(1,j)] \wedge [\bigwedge_{j\leq i} \neg\mathcal{F}\ \varphi(2,j)]$,
  where
  $\varphi(k,j) = \mathcal{P}(\alpha^k, notify_k, j) \wedge [\neg\mathcal{P}(-, update_k, \alpha^k)\ \mathcal{U}$
  $\{\mathcal{P}(j, notify'_k, \alpha^k) \wedge [\neg\mathcal{P}(-, update_k, \alpha^k)\ \mathcal{U}$
  $\mathcal{P}(\alpha^k, notify_k, j)]\}]$.

  Using the reasoning analogical to the one above, we can conclude that $\{\varphi_i\}_{i\in\mathbb{N}_0} \in Property(\mathcal{S}^2\&\mathcal{O}, 1)$

- $\overline{\varphi}_i = \neg\mathcal{F}\ \varphi(\alpha,\beta) \wedge \neg\mathcal{F}\ \varphi(\alpha\alpha,\beta)$.
  We need to compute the value $|\overline{\mathcal{S}^2\&\mathcal{O}}|_X$ for
  $X = \{(-, update_1, \alpha), (\alpha, notify_1, \beta), (\beta, notify'_1, \alpha),$
  $(-, update_2, \alpha\alpha), (\alpha\alpha, notify_2, \beta), (\beta, notify'_2, \alpha\alpha)\}$
  and it holds that
  $|\overline{\mathcal{S}^2\&\mathcal{O}}|_X \leq 1$.

- Then after verifying the models $\mathcal{S}^2\&\mathcal{O}_0$, $\overline{\mathcal{S}^2\&\mathcal{O}}_0$ and $\overline{\mathcal{S}^2\&\mathcal{O}}_1$ we can conclude that the property always holds.

4.3 Closing remarks

We could also study the general issue of whether for a certain property (parametrized with a number of Subjects and

Observers) there exists n_{max} and n'_{max} such that the validity of the property for systems with at most n'_{max} Subjects and at most n_{max} Observers implies the validity of the property on all systems. The reasoning would follow the same intuition as the verification of the properties from the set $Property(\mathcal{D}, m)$ on the dynamic system $\mathcal{D}$. However from space reasons, we do not study this general case here.

5. REFERENCES

[1] L. Brim, I. Černá, P. Vařeková, and B. Zimmerova. Component-Interaction Automata as a Verification-Oriented Component-Based System Specification. In *Proceedings of SAVCBS'05*, pages 31–38, Lisbon, Portugal, September 2005.

[2] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, USA, January 2000.

[3] P. Vařeková, P. Moravec, I. Černá, and B. Zimmerova. Effective-Verification of Systems with a Dynamic-Number of Components. In *Proceedings of SAVCBS'07*, Dubrovnik, Croatia, September 2007.

[4] I. Černá, P. Vařeková, and B. Zimmerova. Component-interaction automata modelling language. Technical Report FIMU-RS-2006-08, Masaryk University, Faculty of Informatics, Brno, Czech Republic, October 2006.

[5] B. Zimmerova, P. Vařeková, N. Beneš, I. Černá, L. Brim, and J. Sochor. *The Common Component Modeling Example: Comparing Software Component Models*, chapter Component-Interaction Automata Approach (CoIn). To appear in LNCS, 2007.

SAVCBS 2007
SHORT PAPERS

Game-Based Safety Checking with Mage

Adam Bakewell
University of Birmingham, UK
a.bakewell@cs.bham.ac.uk

Dan R. Ghica
University of Birmingham, UK
d.r.ghica@cs.bham.ac.uk

ABSTRACT

Mage is a new experimental model checker based on game semantics. It adapts several techniques including lazy (on-the-fly) modelling, symbolic modelling, C.E.G.A.R. and approximated counterexample certification to game models. It demonstrates the potential for truly compositional verification of real software.

Categories and Subject Descriptors

D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*Semantics*

General Terms

Verification

Keywords

Software model checking, game models, symbolic automata, compositional verification, data approximation, refinement

1. GAME-BASED SAFETY CHECKING...

Game Models Intuitively, the game model of a program can be generated by calling the program with every possible combination of arguments; and when the program calls on one of its free identifiers returning every possible result. The game model is then the set of sequences of values passed in and out. Game models have the following key advantages.

1. **Compositionality** The model of a composite program $f(a)$ is obtained by applying a simple 'compose' rule to the models of f and a. Thus components can be modelled and checked independently.

2. **Full abstraction** That is, both soundness (presence of an error-action in the model implies a fault in the program) and completeness (all program faults are present as error-actions in the model).

Sixth International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2007), September 3-4, 2007, Cavtat near Dubrovnik, Croatia.

3. **Black-box** The game model only reports observable actions. This inherent abstraction provides the usual benefits: code privacy, model concision, separate analysis of components.

Safety Checking In game models, program safety reduces to event reachability (i.e. the program passing out an error value or calling an exception). Building an automaton representating the model and searching the transition space of the automaton for error actions implements safety checking.

For simple languages like *Idealized Algol* (IA) [1], which have regular language game models, a finite-state automaton is constructed and a sound and complete safety check by exhaustive search *can* be realized, as in the first game-based model checker which was presented at SAVCBS 2003. The caveat is that for realistic types (e.g. 32-bit integers) the models are often too big, despite the black-box property.

More powerful features like recursive types and higher-order functions need infinite-state automata so state approximation and loss of soundness must be incorporated as usual.

Data Approximation The second games-based model checker, GameChecker [5] used *data approximation* and adapted the *CEGAR* (that is, "counterexample-guided approximation refinement") technique [4] to game models. This allows checking to begin with a very small model and gradually increase the precision of the data types in parts of the program that generate potential counterexamples. The results [6] show another success-in-principle: programs with realistic type signatures can be modelled and checked. But literal interpretation of the game-theoretic approach — build models from component models and pass the final product to a checker — makes analysis of large programs impracticable.

2. ...WITH MAGE

Mage Our new safety checker, Mage[1], makes several advances over the previous state-of-the-art that overcome some of the problems inherent to compositional black box models. These ideas, outlined below, give asymptotic improvements in the complexity of many safety checking problems. These big performance gains have been won by bending and breaking the game approach in various ways.

Lazy Safety Checking Models are big. But this should not be the barrier in safety checking because the result of safety checking is a verdict (and perhaps a counterexample); not a model. Thus, actually building a model then checking

[1] `http://www.cs.bham.ac.uk/~axb/games/mage/`.

it is very space inefficient. It is also very time inefficient if the model contains errors that show up early in the check. Mage generates parts of the model as they are demanded by the checker. And it stops as soon as it detects unsafety.

Symbolic Game Models The game models are regular languages represented as automata. To fit with lazy checking it is much better to implement models in an implicit — rather than constructed — form: an initial state and the next-state function suffice and we call this representation a symbolic model (cf. [2]). Symbolic composition is especially useful as it only considers parts of the model that are demanded by the checker: an integer function f model might have a different behaviour for each of its 2^{32} arguments but only those behaviours demanded by the possible values of the argument a are considered when generating the model of $f(a)$. Thus symbolic models are still *defined* compositionally but the checker can use information about the surrounding context to make a significant efficiency gain when searching the symbolic transitions.

Data Approximation We replaced integers with finite ranges. This breaks the soundness direction of full abstraction so in general only produces possible-counterexamples but can be very effective in eliminating error-free sub programs from the search and quickly detecting data-independent errors.

Approximated Counterexamples Data approximation adds behaviours to the model. Therefore counterexamples must be certified — i.e. is the image, under approximation, of an error in the unapproximated model. Model checkers usually analyse counterexamples with a SAT solver. Mage uses domain-specific knowledge to implement a simpler and more efficient solution: non-determinism on the path through the approximated model to the error indicates a possibly-false counterexample.

CEGAR Finding a possibly-false counterexample causes Mage to refine the re-check the model. Refinement means increasing the precision of the data approximations for those values that led to the counterexample. The symbolic model is refined simply by modifying the type annotation on affected variables. The model-check-certify-refine loop repeats until a true counterexample is found or every possibly-false counterexample is eliminated. Termination is guaranteed because each refinement makes a model strictly less approximate and ultimately the unapproximated models are finite.

Individuated Refinement It is a disadvantage to force different uses of the same variable to share the same approximation: approximate values needed at one site to generate unsafety are then considered at other sites, typically leading to more false counterexamples and more backtracking in the search and more refinement iterations than would otherwise happen. Mage identifies which variable site generated (or consumed) each value in a possible-counterexample. and refines the approximation used at each site individually.

Grey-box models To support the refinement and certification techniques we have to leak some information about internal actions. For certification this creates a constant overhead; for refinement the cost can be larger. So our models are not strictly black-box; merely as black as possible.

stack size	Mage	GameChecker	Blast
2	0.1	10.1	1.6
4	0.1	27.5	3.3
8	0.2	112.6	4.6
16	0.4	780.7	7.8
32	1.2	12,268.1	17.3
64	3.9	over 7 hours	43.7
128	13.9	-	145.3
256	54.8	-	space exhausted

Table 1: Stack overflow detection tests.

3. RESULTS

Stack Verification We compare Mage with the earlier CEGAR game-based checker GameChecker on the same verification problem. We also compare it with the powerful non-game-based model checker, Blast [7] (translating the problem from IA into C makes no semantic difference). Blast is a suitable non-game comparison because it also uses lazy modelling and refinement techniques and it represents the state of the art in verification based on *predicate abstraction* and it can verify significant applications such as device drivers. The problem is to discover contexts that lead to underflows and overflows in a stack of integers where the stack is represented by a finite array and the stack interface presents a push and a pop method that call exceptions when the empty stack is popped or a full stack is pushed.

Overflow Table 1 shows the time taken (on the same machine, in seconds) for the three tools to detect a context leading to an overflow for stacks of different sizes. The Mage times are roughly linear; GameChecker is exponential because it is dominated by model building; Blast is also roughly linear but suffers resource problems with larger stacks. Mage can handle stacks of thousands of elements.

Underflow For the underflow search problem the laziness of both Mage and Blast allow the counterexample "pop empty" to be discovered in a fraction of a second for stacks of billions of elementss. For GameChecker the need to build the model before checking causes similar (slightly faster) results to the overflow problem.

Future Prospects Results such as these suggest that the compositional games approach should be scalable to handle much larger software projects. Our research agenda is to extend the framework to a practical language such as C and then to combine the pure model checking with support from program analysis.

4. REFERENCES

[1] Abramsky, S., Ghica, D.R., Murawski, A.S., Ong, C.H.L.: Applying game semantics to compositional software modeling and verification. In: TACAS. (2004) 421–435

[2] Ball, T., Rajamani, S.K.: Bebop: A symbolic model checker for boolean programs. In: SPIN. (2000) 113–130

[3] Ball, T., Cook, B., Levin, V., Rajamani, S.K.: Slam and static driver verifier: Technology transfer of formal methods inside Microsoft. In: IFM. (2004) 1–20

[4] Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: CAV. (2000) 154–169
[5] Dimovski, A., Ghica, D.R., Lazic, R.: Data-abstraction refinement: A game semantic approach. In: SAS. (2005) 102–117
[6] Dimovski, A., Ghica, D.R., Lazic, R.: A counterexample-guided refinement tool for open procedural programs. In: SPIN. (2006) 288–292
[7] Henzinger, T.A., Jhala, R., Majumdar, R.: The Blast software verification system. In: SPIN. (2005) 25–26

Specification and Verification of Trustworthy Component-Based Real-Time Reactive Systems *

Vasu Alagar and Mubarak Mohammad
Department of Computer Science and Software Engineering
Concordia University
Montreal, Quebec, Canada H3G 1M8
{alagar,ms_moham}@cse.concordia.ca

ABSTRACT

This paper presents a formal methodology for the development of trustworthy real-time reactive systems (RTRS). Safety and security are considered as the two significant properties for trustworthy RTRS. The paper presents an overview of a component-based modeling that involves formal descriptions for trustworthy components. Then, Formal rules are introduced for the automatic generation of behavior protocol based on the formal definitions of trustworthy components. A model-checking method to formally verify security and safety properties in the component model is presented.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements / Specifications—*Methodologies*; D.2.4 [**Software Engineering**]: Software / Program Verification—*Formal methods, Model checking*

General Terms

Design, Security, Verification

Keywords

Trustworthiness, Components, Real-Time Reactive Systems

1. INTRODUCTION

In this paper we explain how trustworthiness can be exploited in the specification and verification of component-based real-time reactive systems (RTRS). In the context of RTRS development we identify *safety* and *security* as the two principal factors contributing to trustworthiness. We propose a verification-oriented design methodology that involves (1) formal specification of component structure and functional/non-functional (trustworthiness) properties, (2) automatic generation of component behavior using the specified structure and restricted by the specified properties, and (3) verification of functional / non-functional component behavior using model checking.

*This research is supported by a Research Grant from Natural Sciences and Engineering Research Council of Canada.(NSERC)

Sixth International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2007), September 3-4, 2007, Cavtat near Dubrovnik, Croatia.

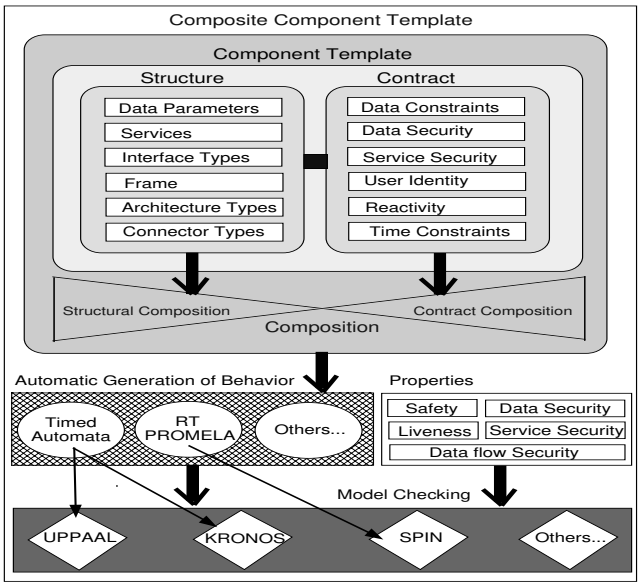


Figure 1: Formal Methodology

Figure 1 depicts our formal methodology. The idea is to formally define the structure of simple components and to define functional and trustworthiness properties at the structural level. There are two benefits for architecting trustworthiness at the structural level. First, it enables the automatic generation of component behavior by analyzing the structure and its properties. Second, it enables reasoning about trustworthiness at the architectural level which is proven to be an important method for attaining trustworthiness [7]. The generated behavior can take different formats depending on defined transformation rules. The defined transformation rules help in (1) automating the process of generating the behavior so that no expertise in behavior specification are required, (2) insuring consistency between the structure and properties defined on one hand and the generated behavior on the other hand, and (3) translating the elements of the structure definitions and the defined properties to a behavior specification format that suits different verification tools. Our goal is to use different verification tools in order to verify a wide range of properties and target different kinds of systems. This is because different verification tools differ in their requirements and abilities [6]: the expressive power of the modeling language, the verification methods used by them, the size and complexity of systems that they can handle, the capabilities suited to different kinds of problems, and the behavior specification format. In this paper we present rules for generating component behavior as ex-

tended timed automata suited for UPPAAL [4] model checker.

In our research, we are focusing on the trustworthiness aspect of RTRS. Reactive systems belong to the class of computer systems that maintain continuous interaction with their environment through stimulus and response. The class of reactive systems in which the response to a stimulus may be strictly regulated by timing constraints is called RTRS. Such systems are required to be trustworthy due to its complexity and the critical contexts it operate in. Although trust is a social concept which is hard to define formally, there is some consensus recently in defining trustworthiness as the degree of user confidence that the system will behave as expected [5]. Safety and security are identified as essential elements for achieving trustworthiness [3]. In the past, research in verifying safety and security properties have progressed in parallel, due to the finding that safety and security can't be formally specified and verified together in any one formal method [8]. We managed to use *component-based development* as a basis for a unified formal model for the specification and verification of safety and security properties of RTRS [3].

Our contributions in this paper are: (1) a formal methodology for developing trustworthy RTRS, (2) transformation rules for the automatic generation of component behavior based on component structure and properties, and (3) model checking safety and security using one method.

2. COMPONENT TEMPLATE - A FORMAL DEFINITION

In our component model, a *component template*, component type, is composed of a *structure* part and a *contract* part. The structure of a template is an abstract external black-box view, called *frame*, and its internal hierarchical structure, called *architecture*. The frame consists of the interface types, the access points to the services provided/requested by the component. Each interface type is associated with a set of services. A service may be parameterized with data parameters. An architecture is a collection of connector types, an abstract view of the tie-ins between interface types. The contract part of the template states the trustworthiness properties required of the system for which the structure is a blue print.

A component is an *instance* of a component template. Every component instantiated from a template has one instance of the structure part defined for the template. The frame of the component is a set of interfaces, where each interface belongs to exactly one interface type in the template frame. It is possible to instantiate multiple interfaces of an interface type. An architecture instance corresponding to a component frame is an instance of the architecture corresponding to the frame in the template, having as many instances of connector types as are required for linking the interfaces in the component. A component's contract constrains the communication pattern at its interfaces and is faithful to the contract part in its template.

In [3] we have introduced a formal component model for trustworthy systems. In this section we present a brief overview of the component model. A component template can be formally specified in terms of its frame and architecture specifications. We focus only on the frame specification because of its relevance to the transformation rules presented in the next section. The internal architecture specification has no impact on the component behavior communicated at the external interfaces.

The frame specification is a tuple $< \Pi, \Sigma, \Lambda, \Xi, \sigma, \Theta, \Gamma, \Omega, \Upsilon, \Psi >$ specifying services, interface types, and properties. The symbol Π denotes a finite non-empty set of interface-types. An *interface* is an instance of an interface type, it inherits the services listed in the type definition. The symbol Σ denotes a finite set of events where each event represents a service provided/requested by the component. The set Σ is divided into a set of input events Σ_{input}, output events Σ_{output}, and internal events $\Sigma_{internal}$ such that $\Sigma = \Sigma_{input} \cup \Sigma_{output} \cup \Sigma_{internal}$ and $\Sigma_{input} \cap \Sigma_{output} \cap \Sigma_{internal} = \emptyset$. An event can carry data parameter values; therefore, we use Λ to denote the finite set of data parameters and define $\Xi : \Sigma \rightarrow \mathbb{P}\Lambda$ as a function that associates with each event a set of data parameters. Events are communicated at the interfaces of the frame; the function $\sigma : \Pi \rightarrow \mathbb{P}\Sigma$ associates a finite non-empty subset of events to each interface-type such that $\forall P, Q \in \Pi, \sigma(P) \cap \sigma(Q) = \emptyset$ i.e. each event is associated with only one interface type. When a request (stimulus) for service is received at an interface, it stimulates the component to perform an action and respond either with an internal processing or with an output event. $\Theta : \Sigma_{input} \rightarrow \Sigma_{output} \cup \Sigma_{internal}$ is a total function that associates a set of possible responses to each request received by the component. The function Θ defines a causality relation between events i.e. $\Theta(e_1) = \{e_2, e_3\}$ means that if event e_1 occurs then event e_2 or e_3 will occur as a response to e_1. The responses of the component can be constrained using (1) time constraints and (2) data parameter constraints. First, Γ denotes the finite set of timing constraints for the events in Σ, where each time constraint involves conjuncts of the form $(t(r) - t(s)) \circ n$, where $t(.)$ is the time function for event occurrences, $s \in \Sigma$ is an input event, $r \in \Sigma$, $r \in \Theta(s)$ is a response to s, $\circ \in \{<, \leq, =, \geq, >\}$, and $n : \mathbb{N}$. Second, Ω denotes a finite set of constraints for the data parameters associated with the events in Σ, where each data constraint of an event $s \in \Sigma$ is a predicate defined over the values of the data parameters $\Xi(s)$ associated with s. If s has n number of responses in $\Theta(s)$ than there must be n number of mutually exclusive data constraints defined over the data parameters of s. This ensures that the responses of s are mutually exclusive. The services provided by the component can be secured and restricted only to authorized users. The introduction of security properties at the frame enriches its behavior by forcing (1) an analysis of the stimulus received before processing it internally, and (2) an analysis of the response before sending it. There are two prerequisites for ensuring security at the interfaces of the component: (1) knowing the identity of the entity on whose behalf the service is requested/provided, henceforth called *user*, and (2) having an explicit definition of an *access control matrix* that defines the *access level* of users to both events and information carried by events. We assume that U denotes the set of users. For the sake of simplicity we assume $AC = \{grant, deny\}$ is the set of access rights for events, and $DA = \{read, write\}$ is the set of allowed actions on data. The function $\Upsilon : U \times \Sigma \rightarrow AC$ defines the event-security access by assigning for every pair $(user, event)$ an authorization which is either $grant$ or $deny$. The function $\Psi : U \times \Lambda \rightarrow \mathbb{P}\ DA$ enforces data-security access. It assigns for every pair $(user, dataparameter)$ an authorization which is a subset of DA. If $\Psi(u, d) = \emptyset$ user u is denied access to data d. The security property is defined in terms of *event-security* and *data-security*. An interface of a component is event-secure if (1) every input event is received from a user who is authorized to trigger the input event, and (2) for every response event sent, the user receiving the response is authorized to view the response. An interface is data-secure if (1) the user has access rights for the data parameters in every stimulus sent by the user, and (2) for every response sent through the interface, the user receiving the response

has access rights for the data parameters in the response.

3. FORMAL VERIFICATION

In this section, we present brief information about UPPAAL model checker. Then, we introduce transformation rules for the automatic generation of component behavior. Finally, we describe how the verification process is conducted using UPPAAL model checker.

3.1 UPPAAL

UPPAAL [4] is a mature model checker that has been used successfully for more than a decade to model check several types of concurrent real time systems. The UPPAAL modeling language is based on timed automata $TA = (L, l_0, K, A, E, I)$ where L is the set of locations denoting states, l_0 is the initial location, K is the set of clocks, A is the set of actions denoting events that cause transitions between locations, E is the set of edges, and I is the set of invariants. Formally, $E \subseteq L \times A \times B(K) \times 2^K \times L$ where $B(K)$ is the set of clock and data constraints denoting guard conditions that restrict transitions, 2^K is the set of clock initializations to set clocks whenever required, and $I : L \rightarrow B(K)$ is a function assigning clock constraints to locations as invariants. UPPAAL extends timed automata with additional features. We present some of those features that are relevant to the this paper:

- **Templates:** Timed automata are defined as templates with optional parameters. Parameters are local variables that are initialized during template instantiation in system declaration.
- **Global variables:** Global variables and user defined functions can be introduced in a global declaration section. Those variables and functions are shared and can be accessed by all templates.
- **Binary synchronization:** Two timed automata can have a synchronized transition on an event when both move to new state at the same time when the event occurs. An event that causes synchronous transition is defined as a *channel*, a UPPAAL data type. A channel can have two directions: input(labeled with ?) and output(labeled with!).
- **Committed Location:** Time is not allowed to pass when the system is in a committed location. If the system state includes a committed location, the next transition must involve an outgoing edge from the committed location.
- **Expressions:** There are three main types of expressions: (1) *Guard* expressions are evaluated to boolean and used to restrict transitions; guard expressions may include clocks and state variables, (2) *Assignment* expressions are used to set values of clocks and variables, and (3) *Invariant* expressions are defined for locations and used to specify conditions that should be always true in a location.
- **Edges:** Edges denote transitions between locations. An edge specification consists of four expressions: *Select:* assigns a value from a given range to a defined variable, *Guard:* an edge is enabled for a location if and only if the guard is evaluated to true, *Synchronization:* specifies the synchronization channel and its direction for an edge, and *Update:* an assignment statements that reset variables and clocks to required values.

In UPPAAL, system properties are expressed formally using a simplified version of CTL [4] as follows:

- **Safety property** is formulated positively stating that some thing good is invariantly true. For example, let φ be a formula, $A\Box\ \varphi$ means that φ should be always true.
- **Liveness property** states that some thing good will eventually happen. For example, $A\diamond\ \varphi$ means that φ will eventually be satisfied.

3.2 Transformation Rules

In this section, we introduce the transformation rules for the automatic generation of component behavior based on the analysis of component's structure and contract defined in the component frame specification. A component-based system is a network of connected components. Every component is mapped to a UPPAAL template in a one to one manner. We assign a parameter to every UPPAAL template to denotes the identifier of the user on whose behalf the component is running. This parameter will be used for ensuring event and data security.

Let $O = \{o_1, \ldots, o_n\}$ be the set of components in a RTRS, $o_i =< \Pi_{I_i}, \Sigma_i, \Lambda_i, \Xi_i, \sigma_i, \Theta_i, \Gamma_i, \Omega_i, \Upsilon_i, \Psi_i >$ such that:$\Sigma_{input} \subseteq \Sigma_i$ denotes the set of stimulus events, $\Sigma_{output} \subseteq \Sigma_i$ denotes the set of output events, $\Sigma_{response} \subseteq \Sigma_{output}$ denotes the set of responses, $\Sigma_{requests} \subseteq \Sigma_i$ denotes the output events sent to other components as requests for services, and $\Sigma_{internal} \subseteq \Sigma_i$ denotes the set of internal events that are local to the component. Let $TA = (L, L_0, K, A, E, I, u)$ be the definition of UPPAAL timed automata where u denotes the user identity parameter associated with the template at its instantiation. Then, the transformation rules construct $T = \{t_1, \ldots, t_n\}$, a set of UPPAAL templates, where t_i is the template constructed from component o_i.

In the definition of a component frame, Π and σ are used in defining the architecture. Therefore, Π and σ don't affect the behavior of the component, hence, are not used in the transformation process. In brief, during the process of constructing $TA = (L, l_0, K, A, E, I)$ from frame specification:

- Σ is used to construct L where every location in L denotes the state of processing an event in Σ,
- Γ is used to construct K and I where a clock in K and an invariant in I are defined for every time constraint in Γ,
- Σ is used to construct A where an action in A is defined for every input or output event in Σ, and
- $\Sigma, \Lambda, \Xi, \Theta, \Omega, \Upsilon$, and Ψ are used to construct E and its associated expressions. More precisely, Λ defines data parameters in Ξ which in turn are used in defining data constraints in Ω that are used along with Υ to define $Guard$ conditions for edges. Σ and Θ are used in defining $Sync$ expression. Ψ is used to control data parameters access in $Update$ expression.

We extend the UPPAAL formal template by adding security features. In the global declaration section, we define: (1) a list of system user identities U, (2) an *event-access control matrix* that defines user access rights to events, (3) a *data-access control matrix* that defines user access rights to events data parameters, (4) an

event security function $EventSecurity: U \times \Sigma \rightarrow boolean$ that searches the event-access control matrix of users-events and returns boolean value indicating whither the user has access or not, (5) a data security function $DataSecurity: U \times \Lambda \rightarrow boolean$ that searches the data-access control matrix of users-data and returns a boolean indicating whither the user has the proper access right (*write* for stimulus parameters and *read* for response parameters) or not.

An informal discussion of the steps for constructing $TA = (L, L_0, K, A, E, I, u)$ is given bellow:

Locations [L]. : A component provides and requests a set of services. The details of service processing are hidden behind component interfaces. Therefore, we use locations to denote the states for processing services. Services are abstracted as events. The function $\Delta: \Sigma \rightarrow L$ constructs for each event a location $\Delta(e)$ in L. The location is the state for processing the event e. The set of locations L can be constructed with the help of Σ as follows:

- [L.1] Create an initial location l_0 to denote the *idle* state where the component is waiting for a stimulus.
- [L.2] Stimulus events correspond to the services provided by the component. For every stimulus event, create a location to represent the service of processing the stimulus.
- [L.3] Output events that are not responses to stimulus correspond to the services requested by the component. For every output event that is not a response to a stimulus create a location.

Clocks [K]. : Time constraints in Γ can be represented by clocks in K and invariants representing clock constraints in I. The set of clocks K can be constructed by creating a clock for every time constraint that constrains the response of a stimulus. Clocks are defined as template's local variables.

Invariants [I]. : Time constraints are defined as location invariants in I. We create an invariant in I for each time constraint in Γ and assign it to $\Delta(e)$.

Actions [A]. : The set of actions A can be constructed by creating an action in A for every input and output event in Σ. Actions are defined as synchronous channels. Input actions are decorated with ? and output actions are decorated with !.

Edges [E]. : The behavior of a component is based on stimuli and responses. Therefore, E can be constructed using Σ according to the rules [E.1], [E.2], and [E.3] defined bellow. The specification of edge expressions is derived from the data parameters Ξ and the constraints Ω, Υ, and Ψ that are related to the action a, which causes the transition, according to the following rules [E.Ex]:

- *Select*: It is used to get a value in a temporary variable for each event data parameter in $\Xi(a)$. These values will be assigned to their corresponding data parameters in the *Update* expression.
- *Guard*: A guard condition is a conjunction $Pr_1 \wedge Pr_2$ such that $Pr_1 \in \Omega$ which is a predicate on data parameters in $\Xi(a)$ and $Pr_2 \in \Upsilon$ which is the event security related to a.
- *Sync*: the action, the event causing the transition.
- *Update*: It includes assignment statements that update data parameters in $\Xi(a)$ and reset the clock in K related to the time constraint in Γ that is defined for a. In order to ensure data security, update statements are constrained by $DataSecurity$ function as follows:
  $\forall d \in \Xi(a),\ d := DataSecurity(u, d)?Select(d) : Null$,
  which means that if the user u has access to the data parameter d then d will be assigned the selected value; otherwise, d will be set to $Null$.

The following rules are used to construct template edges. After constructing each edge, the rules in [E.Ex] are used to define its expressions.

- [E.1] For every stimulus e create an edge from the initial location l_0 to $\Delta(e)$. If $\Theta(e)$ is time constrained then we should reset the clock.
  After finishing the processing of e by sending $\Theta(e)$, the component can go back to idle state waiting for the next stimulus. Therefore, for every response, we create an edge from $\Delta(e)$ back to l_0.
- [E.2] In order to provide the required services, the component may request services from other components. When a stimulus e has a response $\Theta(e) \in \Sigma_{request}$ then create an edge from $\Delta(e)$ to $\Delta(\Theta(e))$ and a second edge from $\Delta(\Theta(e))$ to l_0 .
- [E.3] the component may have a concurrent behavior. It can receive stimuli while processing others. Therefore, we create an edge from every location that represents stimulus processing location l_{p1} to the other stimulus processing locations l_{p2}. Use intermediate committed locations and split the edge into two edges: (1) an edge from l_{p1} to the committed location labeled with the stimulus and (2) an edge from the committed location to l_{p2} labeled with the response of l_{p1}. The reason for having two edges is that UPPAAL doesn't allow having two synchronous channels on an edge.

EXAMPLE 1. *Let* $< \Pi, \Sigma, \Lambda, \Xi, \sigma, \Theta, \Gamma, \Omega, \Upsilon, \Psi >$ *be a frame specification where* $P = \{p_1\}$*;* $\Sigma = \{e_1, e_2, e_3\}$ *such that* $\Sigma_{input} = \{e_1\}$, $\Sigma_{response} = \{e_2\}$, $\Sigma_{request} = \{e_3\}$*;* $\Lambda = \{d\}$*;*$\Xi(e_1) = \{d\}$*;* $\sigma(p_1) = \{e_1, e_2, e_3\}$*;* $\Theta(e_1) = e_2$, $\Theta(e_1) = e_3$*;* $\Omega(e_1, e_2) : d > 10$, $\Omega(e_1, e_3) : d \leq 10$*;* $\Gamma(e_1, \theta(e_1)) = [0, 5]$*;* $U = \{u_1\}$, $\Upsilon(u_1, e_1) = \Upsilon(u_1, e_2) = \Upsilon(u_1, e_3) = grant$, $\Psi(u_1, d) = \{read, write\}$. *Figure 2 shows the extended time automata generated for this example using the transformation rules. The construction is done as follows:*
Locations*:* l_0 *is created according to rule [L.1],* l_1 *according to [L.2], the invariant at* l_1 *according to [I], and* l_2 *according to [L.3].*
Edges*: created according to the following rules and [E.Ex]: (1)* (l_0, e_1, l_1) *is created according to [E.1], (2)* (l_1, e_2, l_0) *is created according to [E.1], (3)* (l_1, e_3, l_3) *is created according to [E.2], and (4)* (l_2, e_3, l_0) *is constructed according to [E.2].*

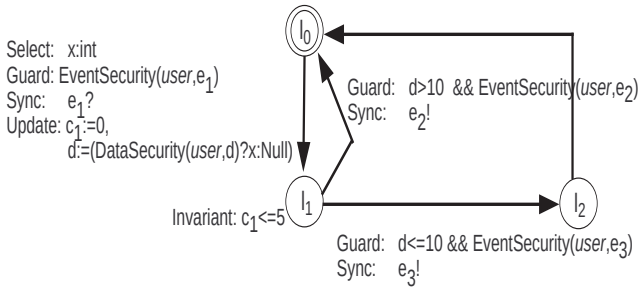


Figure 2: Example

Clocks*: c_1 and the invariant at l_1 are created according to rules [K] and [I].*
Actions*: $e_1?, e_2!, e_3!$ are created according to [A] and [E.Ex]*

3.3 Verification Process

In [2] we have applied the methodology successfully to specify and model check a simplified version of the steam boiler controller case study [1]. The system consists of 3 components: (1) *controller* has 10 locations, (2) *level measuring* has 3 locations, and (3) *monitoring* has 5 locations. The steps of performing the verification process are:

- using UPPAAL *editor*, we specified the components as UPPAAL templates using the automatic transformation rules. Then, in the system declaration section of the editor, we created instances of the templates and defined the RTRS as the parallel composition of the instances,
- using UPPAAL *verifier*, we specified safety, liveness, event security, and data security properties.
  - **Event security**: An event can be triggered only by a user whose access level is *grant*. This is expressed as: $A\Box$ *for all(i:int[1,NoOfUsers]) C.user==i &&* $C.event_x$ *imply EventSecurity(i,*$event_x$*)==grant*. It means: invariantly, in all system executions, $event_x$ can be triggered by authorized users only.
  - **Data security**: A data parameter value should be visible only to authorized users. This is expressed as the invariant: $A\Box$ *for all(i:int[1,NoOfUsers]) C.user==i && DataParameter!=Null imply DataSecurity(i,DataParameter) ==read*. It means: invariantly, in all system executions, the value of $DataParameter$ can be visible only to authorized users; otherwise, it is set to $Null$.
- We executed the model checker to verify the properties against the defined system.

The experiment was performed on two machines: (1) An average PC workstation with 512MB of memory and Pentium IV processor running *Windows XP Home Edition*, and (2) a powerful server with 3GB of memory and Pentium Xeon 3GH running *Windows Server 2003*. Table 1 presents the time duration of model checking each property using the two machines. The time ranges between 1 to 2 minutes on the workstation.

Table 1: Time Duration of Model Checking

Result	workstation	server
Safety	1.49 min	0.12 min
Liveness	1.29 min	0.12 min
Event Security	1.21 min	0.11 min
Data Security	2.06 min	0.12 min

4. CONCLUSION

We have introduced (1) a formal methodology for developing trustworthy systems and (2) formal set of rules for generating the behavior of a component-based model, and (3) model check functional and non-functional properties using UPPAAL model checker. We have applied our method for a simple version of the steam boiler controller problem. We plan to evaluate our method on problems from different domains where both safety and security are critical. Our research directions include: (1) investigating the requirements of an ADL for expressing trustworthiness and (2) building a visual interface tool that enables software architects to specify trustworthy component-based systems. Then, we will derive the formal description automatically from the visual notations and generate system behavior in different formats. The generated behavior will be input into model checkers to perform the verification process. This will hide the complexity of formal specification and enable software architects to easily design and verify trustworthy systems.

5. REFERENCES

[1] Jean-Raymond Abrial, Egon Börger, and Hans Langmaack. Formal methods for industrial applications, specifying and programming the steam boiler control. London, UK, 1996. Springer-Verlag.

[2] Vasu Alagar and Mubarak Mohammad. A formal approach for the development of trustworthy component-based rtrs - case study. http://users.encs.concordia.ca/[tilda]ms_moham/sv.pdf.

[3] Vasu Alagar and Mubarak Mohammad. A component model for trustworthy real-time reactive systems development. In *FACS'07*, Sophia-Antipolis, France, Sept 2007.

[4] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on uppaal. In *Proceedings of SFM-RT'04*, 2004.

[5] Ivica Crnkovic and Magnus Larsson, editors. *building reliable component-based Software Systems*. Artech House, 2002.

[6] John C. Knight Elisabeth A. Strunk, M. Anthony Aiello. A survey of tools for model checking and model-based development. Technical Report CS-2006-17, Dept. of Computer Science, University of Virginia, June 2006.

[7] Cristina Gacek and Rogrio de Lemos. *Structure for Dependability: Computer-Based Systems from an Interdisciplinary Perspective*, chapter Architectural description of dependable software systems, pages 127–142. Springer London, 2006.

[8] John McLean. A general theory of composition for a class of "possibilistic" properties. *IEEE Trans. on Software Engineering*, 22(1):53–67, 1996.

Components, Objects, and Contracts

Olaf Owe
Department of Informatics
University of Oslo, Norway
olaf@ifi.uio.no

Gerardo Schneider
Department of Informatics
University of Oslo, Norway
gerardo@ifi.uio.no

Martin Steffen
Department of Informatics
University of Oslo, Norway
msteffen@ifi.uio.no

ABSTRACT

Being a composite part of a larger system, a crucial feature of a component is its *interface*, as it describes the component's interaction with the rest of the system in an abstract manner. It is now commonly accepted that simple syntactic interfaces are not expressive enough for components, and the trend is towards *behavioral* interfaces.

We propose to go a step further and enhance components with *deontic contracts*, i.e., agreements between two or more components on what they are *obliged, permitted*, and *forbidden* to do when interacting. This way, contracts are modeled after legal contracts from conventional business or judicial arenas. Indeed, our work aims at a framework for *e-contracts*, i.e., "electronic" versions of legal documents describing the parties' respective duties.

We take the object-oriented, concurrent programming language *Creol* as starting point and extend it with a notion of components. We then discuss a framework where components are accompanied by contracts and we sketch some ideas on how analysis of compatibility and compositionality could be done in such a setting.

Categories and Subject Descriptors

D.2.13 [**Software Engineering**]: Reusable Software; D.1.3 [**Programming Techniques**]: Concurrent programming; D.1.5 [**Programming Techniques**]: Object-oriented programming; F.3.1 [**Logics and meanings of programs**]: Specifying and Verifying and Reasoning about Programs

General Terms

Verification

Keywords

Components, compositionality, contracts, interfaces, object-orientation, Creol, deontic logic

1. INTRODUCTION

Sixth International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2007), September 3-4, 2007, Cavtat near Dubrovnik, Croatia.

We propose to combine components with *deontic contracts,* i.e., agreements between two or more components on what they are *obliged, permitted*, and *forbidden* to do when interacting. This way, contracts are modeled after legal contracts from conventional business or judicial arenas. Indeed, our work aims at a framework for *e-contracts*, i.e., "electronic" versions of legal documents describing the parties' respective duties. They go beyond standard behavioral interface descriptions, which typically describe sets of interaction traces. In particular, contracts, in the intended application domain, involve a *deontic* perspective, speaking about obligations, permissions and prohibitions, and also contain clauses on what is to happen in case the contract is not respected. This deontic aspect is typical for natural language legal contracts which we use as a starting point and which we aim to formalize.

The problem

We are concerned with finding a good programming and specification language, and appropriate abstractions for developing components in an integrated manner within the object-oriented paradigm. We are interested in enhancing components with more sophisticated structures than interfaces, targeted towards e-contracts. In that context, we address the following questions.

Design: How to develop components in a programming environment facilitating rapid prototyping and testing?

Composition and compatibility: How do we know that two or more components will not conflict with each other when put together?

Substitutability: How to guarantee that replacing a component will not introduce new unexpected behaviors?

Deontic specification: How to specify what a component is supposed to do, what it may do, and what it should not do?

Contract violation: How to react in case a component does what it is not supposed to do?

These issues are crucial in component-based software development and deployment. In fact, most of the questions, perhaps apart from the deontic aspect, are not new to the component-based software engineering community.

We propose a model combining the following ingredients: 1) As underlying object-oriented language, we use the concurrent language *Creol.* 2) As mentioned, we propose a notion of deontic contract, written in a *contract language.* 3)

The contract is associated with the component model, allowing static and dynamic reasoning on component consistency and conformance, using a *contract logic*. In the following section we discuss some differences between objects and components. In Section 3 we clarify the notion of "contract" used on this paper. In Section 4 we sketch the three ingredients mentioned above, whereas in Section 5 we describe our proposed framework. We conclude in the last section.

2. COMPONENTS VS. OBJECTS

Even if there is no clear-cut definition of what exactly is a component, and what distinguishes the notion from a software module or just an object, we highlight here some essential differences between objects and components.

- Components are supposed to be self-contained units and independently deployable. This is not the case in general for objects, as they usually are not executable by themselves.
- If developed using the object-oriented paradigm, a component may contain many objects which are encapsulated and thus are not accessible from other components. If an object creates another object inside a component, this new object is not visible from the outside unless explicitly allowed by the interface. Objects in most languages do not have this feature.
- Components are static entities representing the main elements of the run-time structure, in contrast to objects, which are dynamic instantiations of classes. A purely class-oriented program does not identify the main elements of a system.[1]

In some sense the above may justify the definition of components as being *just* a collection of "circles" (objects) encapsulated inside a "box", which in turn could also be a kind of object typed by an interface. It is now accepted that such interfaces should not only take into account functional aspects but should take into account the history of interactions, or in other words be *behavioral*.

3. ON THE NOTION OF CONTRACTS

The term "contract" is understood in various ways by different research communities. We briefly recall some of its more common definitions or informal meanings.

1. *Conventional contracts* are legally binding documents, establishing the rights and obligations of different signatories, as in traditional judicial and commercial activities.
2. *Electronic contracts* are machine-oriented and may be written directly in a formal specification language, or translated from a conventional contract. The main feature is the inclusion of certain normative notions such as *obligations*, *permissions*, and *prohibitions*, be it directly or by representing them indirectly. In this context, the signatories of a contract may be objects, agents, web services, etc.
3. Some researchers informally understand contracts as *behavioral interfaces*, which specify the history of interactions between different agents (participants, objects, principals, entities, etc). The rights and obligations are thus determined by legal (sets of) traces.
4. The term "contract" is sometimes used for specifying the interaction between communicating entities (agents, objects, etc). It is common to talk then about a *contractual protocol*.
5. *Programming by contract* or *design by contract* is an influential methodology popularized first in the context of the object-oriented language Eiffel [6]. Contract here means a relation between pre- and post-conditions of routines, method calls, etc.
6. In the context of web services, "contracts" may be understood as a *service-level agreement* usually written in an XML-like language like IBM's Web Service Level Agreement (WSLA [10]).

We are mostly concerned with the first two meanings, though, to be able to reason and operate on contracts, it is natural to have the contracts written in a formal language, and thus the second meaning is more adequate. Obviously, the mentioned interpretations are not absolutely disjoint. The point we like to stress here is the importance of the mentioned normative aspects, which is very typical for (electronic) contracts capturing the spirit in which legal contracts are usually written. Besides those deontic aspects, electronic contacts in our sense also include behavioral aspects (making statements about the order of interactions at the interface), and may also relate the pre- and post-conditions of methods, as in point 5. But what is missing in usual interface and behavioral specifications are linguistic means to make the consequences explicit; e.g. what happens (or should happen) when the normative requirements are violated.

4. COMPONENTS, OBJECTS AND CONTRACTS

Creol

Creol is an object-oriented, concurrent programming and modeling language developed at the University of Oslo. For a deeper coverage of the language, its design and semantics, we refer to the Creol web pages [3] and to [4, 5]. The choice of Creol as underlying language is motivated as follows:

Concurrency: It is a language for open, distributed systems, supporting concurrency and asynchronous method calls. The concurrency model is that of loosely coupled active objects with asynchronous communication. This makes it an attractive basis for component-based systems.

Object-orientation: Creol is an object-oriented, class-based language, with late binding and multiple inheritance. It is strongly typed, supporting subtypes and sub-interfaces.

Interfaces: Creol's notion of *co-interface* allows specification of required and provided interfaces. The language supports behavioral interfaces, based on assume-guarantee specifications expressed in terms of the communication history.

[1] However, early OO languages, including Simula and Beta, had a notion of block prefixing giving rise to static units which resemble components.

Formal foundations: Creol has a formal operational semantics defined in rewriting logic. The core of the language has an operational semantics consisting of only 11 rewrite rules. This makes it easy to extend and modify the language and the semantics. We may reuse the operational semantics when formalizing the extension to components. Based on the formal semantics, the language comes with a simple reasoning system and composition rules.

Tool support: Creol has an executable interpreter defined in the Maude language and rewriting tool. This provides a useful test-bed for the implementation and testing of our component-based extension. The Maude tool may be used for simulation, model checking, and analysis.

Contract language

Formally, we let component interface descriptions be based on the contract language $\mathcal{CL}$ developed in [9]. $\mathcal{CL}$ is a language tailored for electronic contracts (e-contracts) with formal semantics in an extension of the μ-calculus. The language follows an *out-to-do* approach, i.e. where obligations, permissions and prohibitions are applied to actions and not to state-of-affairs. The language avoids the main classical paradoxes of deontic logic, and it is possible to express (conditional) obligations, permissions and prohibitions over concurrent actions keeping their intuitive meaning. Moreover, it is possible to represent (nested) CTDs (*contrary-to-duty*, i.e. what happens when an obligation is not fulfilled) and CTPs (*contrary-to-prohibitions*, i.e which action to be performed in case of violating a prohibition).

Components and Contracts

We list some of the main features of contracts in the context of component-based development and deployment. Contracts associated with components enhance behavioral interfaces and give the following added value:

1. If written in a formal language with formal semantics and proof system, a contract can be proved to be conflict-free, both by model checking and logical deduction techniques. The automatic checks can also reveal incompleteness in the specification, for instance it may indicate that no escalation is agreed upon in case one of the partners acts contrary to its contract.
2. The use of contracts may assist the developer during the development phase to check whether a component may enter into conflict with others, through a static analysis of contract compatibility. The appropriate notion of compatibility in the presence of obligations, permissions, and prohibitions needs to be developed.
3. A well-founded theory of contracts should provide the following kinds of analysis:
   - Determine whether a contract is *covered* by another one, i.e. a well-defined notion of sub-contract. This will help deciding whether a component may be replaced by another one in a safe manner.
   - Allow decisions on whether paying a penalty in case of one contract violation is beneficial or not when sub-contracting. Assume component A has a contract with component B where it is stipulated that A must "pay" x to B in case of contract violation. Suppose now that such violation depends on a service provided by C to A and that there is a contract between A and C stating that C must pay y to A in case of their own contract violation. Then a theory of contracts would allow A to determine whether it is good to compose with B. During the development phase this kind of information may help defining sub-contracting which are not against a component's own interest.
   - A negotiation phase could be added prior to the composition of two or more components. In this phase a contract could be negotiated before the final signature, as in the context of web services.
4. A run-time contract monitor will guarantee that the contract is respected, including the penalties and escalations in case of contract violation (CTDs and CTPs). We expect such a monitor could be extracted from the components contracts in a (semi-)automatic way.

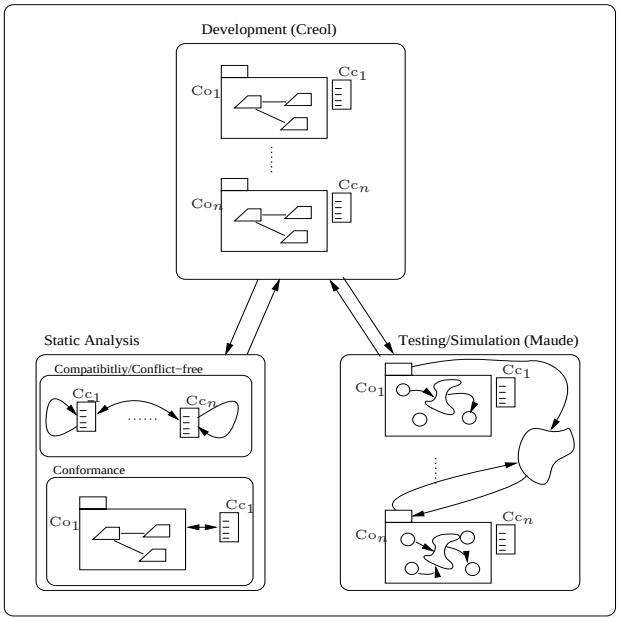


Figure 1: Development phase.

5. PROPOSED FRAMEWORK

The logical semantics of $\mathcal{CL}$ opens the way to use the logic proof system of μ-calculus, as well as existing model checkers. Initial work on model checking a contract has been presented in [8]. The combination of components, objects and contracts may be done as sketched in our proposed framework, involving both the component's development and deployment phase, using Creol as the development platform.

Development Phase.

During this phase our framework may be summarized as follows (see Fig. 1):

Development: Each component has associated one or more contracts in the sense discussed above, i.e., specifying the obligations, permissions, and prohibitions in the component's interacting behavior.

Static Analysis: Before deployment, the contract is formally analyzed to guarantee that it is contradiction

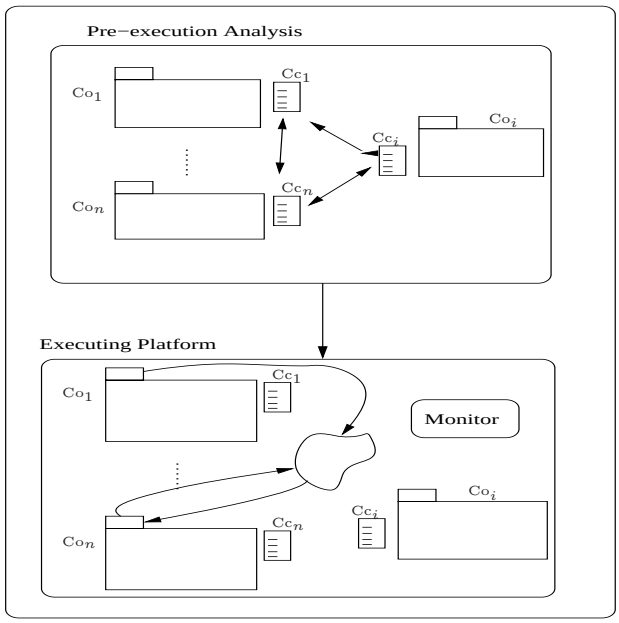


Figure 2: Deployment phase.

free. This might be done by using a proof system or by model checking. Static conformance between the component and its contract is also proved.

Testing/Simulation: Static analysis techniques cannot validate every aspect of a system. Testing and simulation are thus needed to complement the above. Since Creol has a formal semantics in rewriting logic, we propose to use the Maude environment to simulate and test each component separately and its interaction with other components being developed.

Deployment Phase.

After the component is released there is still no complete guarantee of it being well suited for the yet unknown platform where it will be executed. We propose the following framework to increase confidence on the component's compatibility with its future environment. See Fig. 2.

Pre-execution Analysis: Before adding a new component to an existing context of other components, the corresponding contracts are checked to guarantee compatibility. If there are disagreements, a phase of negotiation may start, or the component is simply rejected. This phase may be considered as a kind of static analysis on the side of the execution platform.

Execution: If the component is accepted after the analysis of the previous phase, then it is deployed. A contract monitor is launched to guarantee that the components behave according to the contracts. In case of contract violation, the monitor must take the corresponding action as stipulated in the contract for such situation, or cancel the contract and disable the component.

6. FINAL DISCUSSION

In this paper we sketched how to enhance components with contracts as complementary to the latest ideas of using behavioral interfaces. In our opinion this approach would benefit from the fact that such contracts could be analyzed logically and model checked in order to find (local) inconsistencies, they could be negotiated and monitored. We believe component-based development and engineering will in some sense be reduced to the same kind of problems one finds in web services and other application domains where contracts are being studied.

The extension of Creol with primitives to define components is not difficult to do as most of the basic constructs are already defined in the language. For instance, contracts might be included as data-types in the language.

The successful use of contracts as we have proposed depends very much on the existence of a suitable formal contract language. We intend to further explore $\mathcal{CL}$ and its semantics to be used in this context. We expect to benefit from its formal semantics in the μ-calculus to further develop proof systems and to explore the possibility of use existing model checking tools.

Though we believe the first phase of the deployment phase could be achieved relatively easy, we are aware that obtaining a contract monitor, when executing a component could represent a big challenge if we intend to do so in real-time. We do not have a solution yet. A very interesting research direction would be to study how to combine meta-programming (e.g. in a reflective language) techniques with a formal (logical) framework for extracting a monitor from one or more contracts.

Related work and further details may be found in the accompanying technical report [7], representing the full version of the paper.

Acknowledgment. This work is partially supported by the Nordunet 3 project *Contract-Oriented Software Development for Internet Services* [1] and the EU-project *Credo, A formal framework for reflective component modeling* [2]. Marcel Kyas as well as the referees have contributed with valuable comments.

7. REFERENCES

[1] COSDIS. `www.ifi.uio.no/~gerardo/nordunet3`, 2007.
[2] Credo. `www.cwi.nl/projects/credo/`, 2006.
[3] Creol. `www.ifi.uio.no/~creol`, 2007.
[4] E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. In *Proc. 2nd Intl. Conf. on Software Engineering and Formal Methods (SEFM'04)*, pages 188–197. IEEE Computer Society Press, Sept. 2004.
[5] E. B. Johnsen, O. Owe, and I. C. Yu. Creol: A type-safe object-oriented model for distributed concurrent systems. *TCS*, 365(1–2):23–66, Nov. 2006.
[6] B. Meyer. *Eiffel: The Language*. Prentice Hall, 1992.
[7] O. Owe, G. Schneider, and M. Steffen. Components, objects, and contracts. Technical Report 363, Dept. of Informatics, Univ. of Oslo, Norway, August 2007.
[8] G. Pace, C. Prisacariu, and G. Schneider. Model checking contracts — a case study. In *ATVA'07*, volume 4762 of *LNCS*, pages 82–97, 2007.
[9] C. Prisacariu and G. Schneider. A formal language for electronic contracts. In *FMOODS'07*, volume 4468 of *LNCS*, pages 174–189, 2007.
[10] WSLA. `www.research.ibm.com/wsla/`.

Compositional Failure-based Semantic Equivalences for Reo Specifications

Mohammad Izadi
Department of Computer Engineering
sharif University of Technology
Tehran, Iran
izadi@ce.sharif.edu

Ali Movaghar
Department of Computer Engineering
sharif University of Technology
Tehran, Iran
movaghar@sharif.edu

ABSTRACT

Reo is a coordination language for modeling component connectors of component-based computing systems. We show that the failure-based equivalences NDFD and CFFD are congruences with respect to composition operators of Reo.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Languages*; F.4.3 [**Mathematical Logic and Formal Languages**]: Formal Languages—*Verification*; D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*Semantics*

General Terms

Languages, Semantics, Verification.

Keywords

Reo Specification Language, Constraint Automata, Failure-based Equivalences, Coordination, Component-based Systems, Semantics.

1. EXTENDED ABSTRACT

The concept of *component based systems*, especially component based software, is a philosophy or way of thinking to deal with the complexity in designing large scale computing systems. One of the main goals of this approach is to compose reusable components by some glue codes. The model or the way in which these components are composed is called *coordination model*. Sometimes there are some formal or programming languages which are used for specification of coordination models. Such languages are called as *coordination languages*. Reo, as one of the most recently proposed coordination languages, is a channel based exogenous coordination language in which complex coordinators are compositionally built out of simpler ones [1, 2, 3]. By using Reo specifications, complex component connectors can be organized in a network of channels and build in a compositional manner. Reo relies on a very liberal and simple notion of channels and can model any kind of peer-to-peer communication. The channels used in Reo networks can be considered as simple communicating processes and the only requirements for them are that channels should have two ends (or I/O interfaces), declared to be *sink* or *source* ends, and a user-defined semantics. At source ends data items enter the channel by performing corresponding write operations. Data items are received from a channel at sink ends by performing corresponding read operations. Reo allows for an open ended set of channel types with user defined semantics.

If we want to be able to reason about properties of specifications or verify their correctness, Reo, as well as any other process specification languages, should be given abstract semantics. The key question in giving a semantic model to a specification language is: "Whenever can we say that two specifications or two models are equivalent?" Numerous definitions of different equivalence-relations for transition system based models have been presented in the literature. *Trace equivalence* (automata-theoretic equivalence), *weak bisimilarity* presented by Milner [9] and *failure-based equivalences* (CSP-like equivalences) such as the equivalence presented by Hoare [7] are examples of these equivalences.

Constraint automaton, as an extension of finite or Büchi automaton, is a formalism proposed to capture the operational semantics of Reo [4]. In a constraint automaton, contrary to finite automata and labeled transition systems, the label of a transition is not a simple character or action name. A transition label contains a set of names and a (constraint) proposition. The set of names indicates the names of ports which are participant in doing the transition and the proposition expresses some constraint about the data of the ports.

In this presentation, we are interested to investigate failure-based equivalences for constraint automata as the abstract semantics of Reo and their congruency with respect to composition operators which are useful in composing Reo specifications. The ultimate goal is to prepare an environment for compositional model checking of Reo specifications using equivalence based reduction method. In this method, the models of components and connectors of a component-based system are reduced with respect to an equivalence relation before building the model of the complete system [5, 6]. An equivalence relation should have two properties in order to be useful in the equivalence based compositional reduction method: it should *preserve* the class of properties to be ver-

Sixth International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2007), September 3-4, 2007, Cavtat near Dubrovnik, Croatia.

ified and also, it should be a *congruence* with respect to the syntactic operators which are used for composing of the components of the model. By congruence relation we mean that the replacement of a component of a model by an equivalent one should always yield a model which is equivalent with the original one. Fortunately, in the context of compositional failure based semantic models of process description languages such as CCS and LOTOS, there are two equivalence relations, called CFFD and NDFD, which have the preservation property: CFFD-equivalence preserves that fragment of linear time temporal logic which has no next-time operator and has an extra operator distinguishing deadlocks [10, 11] and NDFD-equivalence preserves linear time temporal logic without next-time operator [8]. It was also shown that CFFD and NDFD are the minimal equivalences preserving the above mentioned fragments of linear time temporal logic.

Now, we introduce an extended definition of constraint automaton by which not only the connectors but also the components can be modeled.

DEFINITION 1. *Let N be a set of port names and Data be a set of data. A* data constraint *g over names set N and data set Data is a proposition, which can be constructed by using the following grammar:*
$g ::= \text{true} \mid \mathrm{d}_A = d \mid g_1 \vee g_2 \mid \neg g \qquad d \in Data, \quad A \in N$
We use $DC(N, Data)$ as the set of all data constraints over names set N and data set Data.
A Constraint automaton *over data set Data is a quadruple $C = (Q, Nam, T, q_0)$ where, Q is a finite set of states, Nam is a finite set of names, such that, $\tau \notin Nam$, $T \subseteq Q \times ((2^{Nam} \times DC(Nam, Data)) \cup \{\tau\}) \times Q$, and $q_0 \in Q$ is the initial state.*
For each $(p, N, g, q) \in T$, it is required that, $N \neq \emptyset$ and $g \in DC(N, Data)$.

The main difference of our definition of constraint automaton and its original definition (defined in [4]) is that in our definition, τ-transitions are permitted, while in its original definition it is not. We use τ-transitions because τ can be used as a symbol for each kind of internal action which is occurred in an actual system but its real type is not important in the modeling process. Thus, by using this kind of constraint automaton, not only the observational behavior of connectors, but also all internal and observable behavior of components can be modeled. Also note that, in principle, a *hiding* operator can hide all port-names of a transition. In such cases, we replace the transition label by τ. Our definition of Constraint automaton is departed from the original one by dropping the requirement that all runs have to be infinite. We also deal with finite runs, which are necessary to argue about deadlock configurations.

Now, we introduce two new composition operators for constraint automata: join (production) of two automata with respect to their common port names and hiding of a port name in all transition labels of an automaton.

DEFINITION 2. *Let $C_1 = (Q_1, Nam_1, T_1, q_{01})$ and $C_2 = (Q_2, Nam_2, T_2, q_{02})$ be two Constraint automata. The* product (join) Constraint automaton *of C_1 and C_2 is:*
$C_1 \bowtie C_2 = (Q_1 \times Q_2, Nam_1 \cup Nam_2, T, q_{01} \times q_{02})$ in which,
1) If $(q_1, N_1, g_1, p_1) \in T_1$ and $(q_2, N_2, g_2, p_2) \in T_2$ and $N_1 \cap Nam_2 = N_2 \cap Nam_1$, then,
$(< q_1, q_2 >, N_1 \cup N_2, g_1 \wedge g_2, < p_1, p_2 >) \in T$,
2) If $(q, N, g, p) \in T_1$ and $N \cap Nam_2 = \emptyset$, then,
$(< q, q' >, N, g, < p, q' >) \in T$,
3) If $(q, N, g, p) \in T_2$ and $N \cap Nam_1 = \emptyset$, then,
$(< q', q >, N, g, < q', p >) \in T$,
4) If $(q, \tau, p) \in T_1$ then, $(< q, q' >, \tau, < p, q' >) \in T$,
5) If $(q, \tau, p) \in T_2$ then, $(< q', q >, \tau, < q', p >) \in T$.
Let $C = (Q, Nam, T, q_0)$ be a Constraint automaton and B be a name, $B \in Nam$. The Constraint automaton resulted by hiding *of B in A is $\exists B[C] = (Q, Nam \backslash \{B\}, T_{\exists B}, q_0)$ where,*
(1) If $(q, \{B\}, g, p) \in T$ then, $(q, \tau, p) \in T_{\exists B}$.
(2) If $(q, N, g, p) \in T$ and $N \backslash \{B\} \neq \emptyset$ then $(q, N \backslash \{B\}, \exists B[g], p) \in T_{\exists B}$, where $\exists B[g] = \vee_{d \in Data} g[d_B/d]$.
(3) If $(q, \tau, p) \in T$ then, $(q, \tau, p) \in T_{\exists B}$.

Now, we can show that failure-based equivalences CFFD and NDFD are congruence with respect to join and hiding operators of constraint automata.

THEOREM 1. *NDFD and CFFD-equivalences are congruences with respect to the product (join) and hiding operators defined for finite constraint automata.*

Based on these congruency results and because of the linear time temporal logic preservation properties of CFFD and NDFD equivalences and their minimality properties (proved in [8]), they will be useful candidates for compositional reduction of models in the process of verifying the properties of component based systems, which their connectors are specified by Reo.

2. REFERENCES

[1] Arbab F., *Reo: A Channel-based Coordination Model for Component Composition*, Math. Struc. in Computer Science, **14(3)**, (2004), 329-366.

[2] Arbab F., *Abstract Behaiviour Types: A foundation model for components and their composition*, science of Computer Programming, **55**, (2005), 3-52.

[3] Arbab F., Mavadat F., *Coordination Through Channel Composition*, Proceedings of Coordination Languages and Models 2002, LNCS, **2315**, Springer-Verlag, (2002).

[4] Baier C., Sirjani M., Arbab F., Rutten J., *Modelling Component connectors in Reo by Constraint Automata*, Science of Computer Programming, **61**, (2006), 75-113.

[5] Clarke E., Long D., McMillan K., *Compositional Model Checking*, Proc. of 4th IEEE Symp. on Logic in Computer Science, (1989), 353-362.

[6] Graf S., Steffen B., *Compositional Minimization of Finite-State Systems*, Proc. of CAV'90, Springer, (1991), 186-196.

[7] Hoare C.A.R., "Communicating Sequential Processes", Prentice-hall, (1985).

[8] Kaivola R., Valmari, A., *The Weakest Semantic Equivalence Preserving Nexttime-less Linear Temporal Logic*, LNCS **630**, Springer-Verlag, (1992), 207-221.

[9] Milner R., "Communication and Concurrency", Prentice-Hall, (1989).

[10] Valmari A., Tienari M., *An Improved Failure Equivalence for Finite State Systems with a Reduction Algorithm*, "Protocol Specification, Testing and Verification", **XI**, (1991), 3-18.

[11] Valmari A., Tienari M., *Compositional Failure Based Semantic Models for Basic LOTOS*, Formal Aspects of Computing **7**, (1995), 440-468.

A Concept for Dynamic Wiring of Components

Correctness in Dynamic Adaptive Systems

Dirk Niebuhr
Clausthal University of Technology
P.O. Box 1253
38670 Clausthal-Zellerfeld, Germany
dirk.niebuhr@tu-clausthal.de

Andreas Rausch
Clausthal University of Technology
P.O. Box 1253
38670 Clausthal-Zellerfeld, Germany
andreas.rausch@tu-clausthal.de

ABSTRACT

Component-based Systems in our days tend to be more and more dynamic. Due to the increased mobility of devices hosting components, components have to be attached or detached to respectively from a system at runtime. This dynamic adaptation of the system configuration imposes several correctness issues. In general it is not possible to determine a correct system configuration without wiring and executing the system in advance. We will discuss approaches how to improve this situation. Finally we will focus on our favorite approach based on runtime testing.

Categories and Subject Descriptors

D.2.4 [**Software**]: Software Engineering—*Software/Program Verification*; D.2.11 [**Software**]: Software Engineering—*Software Architectures*; F.3.1 [**Theory of Computation**]: Logics and Meanings of Programs—*Specifying and Verifying and Reasoning about Programs*

General Terms

Design, Reliability, Verification

Keywords

Dynamic Adaptive Systems, Reconfiguration, Runtime Testing, Correctness, Component, Adaptation

1. INTRODUCTION

To produce systems out of *IT components* component-based development approaches have been developed and successfully applied over the past years changing the predominant development paradigm: Systems are no longer redeveloped from scratch, but composed of existing components [4, 1]. Nowadays, these IT components are being more and more used within an organically grown, heterogeneous, and dynamic IT environment. Users expect these IT components to collaborate autonomously with each other and provide a real added value to the user. On the other hand, we depend more and more on these organically grown IT systems. Hence their correctness has to be guaranteed even though these systems are never developed and tested in advance. These *dependable adaptive IT systems* need to have the ability to dynamically attach and detach dynamic adaptive IT components during runtime. Moreover they need to detect and avoid possible resulting incorrect system configurations during runtime.

In this paper we present our approach of achieving runtime dependability of these systems by runtime testing. We will sketch the proposed runtime testing approach illustrated by a small example in Section 3.

Sixth International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2007), September 3-4, 2007, Cavtat near Dubrovnik, Croatia.

2. DYNAMIC WIRING OF COMPONENTS

Imagine a very simple system containing three components *ComponentA*, *ComponentB*, and *ComponentC*. *ComponentA requires* a component providing *InterfaceA* whereas *ComponentB* respectively *ComponentC provide InterfaceB* respectively *InterfaceC*. Moreover each of the interfaces comes along with it's own specification (t_A, t_B, t_C) of required respectively assured properties. This component landscape is depicted in Figure 1.

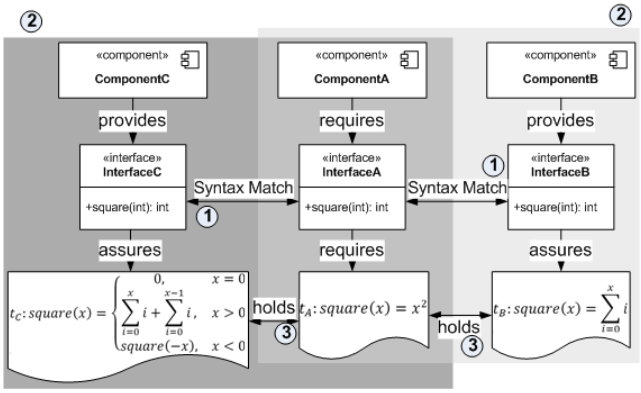


Figure 1: The Artifacts of the Example System

During configuration we have to decide at runtime, whether *holds(prov, req)* is valid for a specific configuration in general, which is depicted in number 3 of Figure 1. This predicate is valid, if the specification of the provided interface implies the specification of the required interface which means proving the refinement relation or showing the equivalence of two Turing machines. This requires the evaluation of implications using second-order logic which refers to the

decision problem. This has been proven to be not decidable by Turing and Church in 1936 [5, 2]. Therefore proving the correctness of a component wiring at runtime in general is not possible. There are several approaches to provide (limited) statements regarding the correctness of the wiring:

1. Using a specification language like regular expressions or finite state machines, which is more restricted and not as powerful in order to get a calculable holds-predicate. In this case you will have to answer the question, whether this specification language is still capable of specifying the desired dynamic adaptive systems respectively you get a less valuable proof in case of an incomplete specification.

2. Checking the correct wiring of components by bisimulation [3]. In this case you would need to compare the states of two simulated system executions *for every system execution step*: one system is containing the requiring component and performs changes to its system state as specified in the required interface specification, the other one is containing the providing component and performs changes to its system state as specified in the provided interface specification. In this case you need to argue, whether the performance of a system using this approach would still be sufficient due to the massive simulation. In addition you only get a proof of correctness for the following execution step.

3. Performing runtime testing during the reconfiguration process: whenever two components should be wired together, test cases are executed within a testbed which check, whether they fit together. In this case you need to show, that the test cases executed during reconfiguration are good enough to expose mismatches of components. Moreover you have to argue, that the testbed is a sufficient representation of the real system environment.

3. RUNTIME TESTING APPROACH

Since we don't want to restrict the specification language and want to retain a good system performance, our approach is using runtime testing in a testbed during the reconfiguration of a system. This reconfiguration may occur, whenever a component appears within a system or a component disappears or fails. In general we use a three-step process. First of all we derive an ordered set of valid system configurations from the set of available components. Then we wire these system configurations within a testbed and check whether all test cases pass. Finally, we transfer this configuration to the production system. We will describe these steps shortly based on an example system.

As you can see, *InterfaceA* and *InterfaceB* respectively *InterfaceA* and *InterfaceC* match syntactically, since they provide syntactically identical methods[1]. This is checked by the syntax match depicted in number 1 of Figure 1. Based on this, two valid system configurations are identified: C_1 wiring *ComponentA* and *ComponentB* and C_2 wiring *ComponentA* and *ComponentC*. The subsets of involved components in these configurations are depicted in number 2 of Figure 1 and are ordered in a way preferring C_1.

We need to test each of these configurations in a testbed. Therefore we duplicate the components, wire the duplicates together and execute test cases. These test cases can be brought by the component user (here: *ComponentA*) since he knows the usage scenarios for the used interface (here: *InterfaceA*) best. They could be provided by the used component as well. A third option would be to generate test cases from the interface specifications (here: t_A, t_B, t_C) of one (or both) of the components. For simplicity we assume, that *ComponentA* provides a test case containing three method calls: $square(0) : 0$, $square(3) : 9$, and $square(-3) : 9$. Within the testbed this test case is executed.

First of all, a duplicate of *ComponentB* is wired together with a duplicate of *ComponentA*. When executing the second method call, the test case fails, since $square(3) = 6$ which contradicts the expected result of 9. Therefore this configuration is marked as invalid. When executing the test cases on the second system configuration, which wires a duplicate of *ComponentA* and a duplicate of *ComponentC*, all test cases pass and therefore this configuration is established in the following. If we want to assure, that the used component behaves as required during execution, we can ensure this as well by additionally checking this after each method call during the system execution in the following. This would cause a large overhead during system execution and therefore may not be applicable for all types of systems. This would correspond to the bisimulation approach.

[1]If components should be wired though their interface methods are not syntactically equal, one could use ontologies.

4. CONCLUSIONS AND FURTHER WORK

Reconfiguration, which means changing the component wiring, is necessary for dynamic adaptive systems since components may enter or leave a system at runtime. However proving the correctness of a component wiring at runtime is not possible in general. Our approach is based on runtime testing component duplicates in a testbed during reconfiguration. This enables us to recognize semantical mismatches of provided and required interfaces at runtime. Therefore we can mark system configurations, wiring these incompatible interfaces, as invalid and chose a valid configuration instead. However we did not take care about cyclic dependencies of components, where an interface provided by *ComponentA* is required by *ComponentB* and vice versa. Moreover we need to investigate test case generation, to enable component developers to provide a single specification of their components and assure good test cases. Moreover we need to check, whether it is sufficient, to execute only test cases involving newly introduced components during reconfiguration.

5. REFERENCES

[1] K. Bergner, A. Rausch, M. Sihling, and A. Vilbig. Putting the parts together – concepts, description techniques, and development process for componentware. In *HICSS 33, Proceedings of the Thirty-Third Annual Hawaii International Conference on System Sciences*. IEEE Computer Society, Jan 2000.

[2] A. Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58:345–363, 1936.

[3] E. Estévez and P. R. Fillottrani. Bisimulation for component-based development. *Journal of Computer Science & Technology*, 1(6), May 2002.

[4] C. Szyperski. *Component Software*. Addison Wesley Publishing Company, 2002.

[5] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. In *Proceedings of the London Mathematical Society*, pages 230–265, 1936.