Karlsruher Institut für Technologie

Karlsruhe Reports in Informatics 2010,13
Edited by Karlsruhe Institute of Technology,
Faculty of Informatics
ISSN 2190-4782

Formal Verification of Object-Oriented Software

Papers presented at the International Conference,
June 28-30, 2010, Paris, France

Bernhard Beckert • Claude Marché (Eds.)

2010

KIT – University of the State of Baden-Wuerttemberg and National Research Center of the Helmholtz Association

Bernhard Beckert • Claude Marché (Eds.)

Formal Verification of Object-Oriented Software

Papers presented at the International Conference,
June 28-30, 2010, Paris, France

Published as:
Technical Report,
Department of Informatics,
Karlsruhe Institute of Technology,
2010-13

Editors

Bernhard Beckert
Karlsruhe Institute of Technology
Institute for Theoretical Informatics
Am Fasanengarten 5, 76131 Karlsruhe, Germany
Email: beckert@kit.edu

Claude Marché
INRIA Saclay - Île-de-France
Parc Orsay Université
4, rue Jacques Monod, F-91893 Orsay Cedex, France
Email: Claude.Marche@inria.fr

Preface

This volume contains the invited papers, research papers, system descriptions, case studies, and position papers presented at the *International Conference on Formal Verification of Object-Oriented Software* (FoVeOOS 2010), that was held June 28–30, 2010 in Paris, France. Post-conference proceedings with revised versions of selected papers will be published within Springer's *Lecture Notes in Computer Science* series after the conference.

Formal software verification has outgrown the area of academic case studies, and industry is showing serious interest. The logical next goal is the verification of industrial software products. Most programming languages used in industrial practice are object-oriented, e.g. Java, C++, or C#. FoVeOOS 2010 aimed to foster collaboration and interactions among researchers in this area.

FoVeOOS was organised by COST Action IC0701 (`www.cost-ic0701.org`), but it went beyond the framework of this action. The conference was open to the whole scientific community. All submitted papers were peer-reviewed, and of the 35 submissions, the Programme Committee selected 23 for presentation at the conference.

We wish to sincerely thank all the authors who submitted their work for consideration. And we would like to thank the Program Committee members as well as additional referees for their great effort and professional work in the review and selection process. Their names are listed on the following pages.

In addition to the contributed papers, the programme of FoVeOOS 2010 included three excellent keynote talks. We are grateful to June Andronick (NICTA, Sydney, Australia), Kim G. Larsen (Aalborg University, Denmark), Francesco Logozzo (Microsoft Research, Redmond, USA) for accepting the invitation to address the conference.

It was a team effort that made the conference so successful. We particularly thank Sarah Grebing, Vladimir Klebanov, and Emmanuelle Perrot for their hard work and help in making the conference a success. In addition, we gratefully acknowledge the generous support of COST Action IC0701, Microsoft Research Redmond, the Institut National de Recherche en Informatique et Automatique (INRIA), and the Karlsruhe Institute of Technology

June 2010

Bernhard Beckert
Claude Marché

Program Committee

Gilles Barthe	IMDEA Software, Madrid, Spain
Bernhard Beckert	Karlsruhe Institute of Technology, Germany
Einar Broch Johnsen	University of Oslo, Norway
Gabriel Ciobanu	University Alexandru Ioan Cuza, Romania
Dave Clarke	Katholieke University Leuven, Belgium
Mads Dam	KTH Stockholm, Sweden
Ferruccio Damiani	University of Torino, Italy
Sophia Drossopoulou	Imperial College, UK
Paola Giannini	University Piemonte Orientale, Italy
Dilian Gurov	KTH Stockholm, Sweden
Reiner Hähnle	Chalmers University of Technology, Gothenburg, Sweden
Marieke Huisman	University of Twente, The Netherlands
Thomas Jensen	IRISA/CNRS, France
Joe Kiniry	ITU Copenhagen, Denmark
Viktor Kuncak	EPF Lausanne, Switzerland
Dorel Lucanu	University Alexandru Ioan Cuza, Romania
María del Mar Gallardo	University of Malaga, Spain
Claude Marché	INRIA Saclay-Île-de-France, France
Julio Mariño	University Politecnica de Madrid, Spain
Marius Minea	Politehnica University of Timisoara, Romania
Anders Møller	University Aarhus, Denmark
Rosemary Monahan	NUI Maynooth, Ireland
Wojciech Mostowski	University Nijmegen, The Netherlands
Peter Müller	ETH Zrich, Switzerland
James Noble	Victoria University of Wellington, New Zealand
Olaf Owe	University of Oslo, Norway
Ernesto Pimentel Sánchez	University of Málaga, Spain
Arnd Poetzsch-Heffter	University of Kaiserslautern, Germany
Erik Poll	University of Nijmegen, The Netherlands
António Ravara	New University of Lisbon, Portugal
Wolfgang Reif	University of Augsburg, Germany
René Rydhof Hansen	University of Aalborg, Denmark
Peter H. Schmitt	Karlsruhe Institute of Technology, Germany
Aleksy Schubert	University of Warsaw, Poland
Gheorghe Stefanescu	University of Bucharest, Romania
Bent Thomsen	University of Aalborg, Denmark
Shmuel Tyszberowicz	University of Tel Aviv, Israel
Tarmo Uustalu	Institute of Cybernetics, Tallinn, Estonia
Burkhart Wolff	University Paris-Sud (Orsay), France
Elena Zucca	University of Genova, Italy

Program Co-Chairs

Bernhard Beckert	Karlsruhe Institute of Technology, Germany
Claude Marché	INRIA Saclay-Île-de-France, France

Organising Committee

Claude Marché *(chair)*	INRIA Saclay-Île-de-France, France
Bernhard Beckert	Karlsruhe Institute of Technology, Germany
Vladimir Klebanov	Karlsruhe Institute of Technology, Germany
Emmanuelle Perrot	INRIA Saclay-Île-de-France, France

Sponsoring Institutions

COST Action IC0701 "Formal Verification of Object-Oriented Software"
Institut National de Recherche en Informatique et Automatique (INRIA)
Karlsruhe Institute of Technology
Microsoft Research

Additional Referees

Davide Ancona	Christoph Feller	Gerhard Schellhorn
Mohamed Faouzi Atig	Pietro Ferrara	Martin Steffen
Viviana Bono	Kathrin Geilmann	Kurt Stenzel
Daniel Bruns	Christoph Gladisch	Volker Stolz
Richard Bubel	Clément Hurlin	Mark Timmer
Jacek Chrzaszcz	Ioannis Kassios	Bogdan Tofan
João Costa Seco	Ilham Kurnia	Varmo Vene
Delphine Demange	Laurent Mauborgne	Amiram Yehudai
Johan Dovland	Keiko Nakata	Greta Yorsh
David Faitelson	Mads Chr. Olesen	

Table of Contents

Abstracts of Invited Talks

Contributed Papers

Timing Analysis of Embedded Software Systems

Kim G. Larsen

Department of Computer Science,
Aalborg University,
kgl@cs.aau.dk

Embedded software is often applied in safety-criticial systems, e.g. the braking system of a car or the steering gear of an airplane. Many of these safety-critical systems are also time-critical, meaning that the calculations performed by the tasks of the embedded system need not only be functionally correct but must be carried out in a timely fashion. In this talk we show how real-time model checking using the verification tool UPPAAL (www.uppaal.com) may be used to give such timing guarantees.

In particular, real-time model checking may be used for efficient schedulability analysis of tasks providing a less pessimistic and more general analysis compared with classical scheduling methods for single-processor. We apply the method to the schedulability analysis of Safety Critical Hard Real-Time Java programs, based on a translation of programs, written in the Safety Critical Java profile to timed automata models verifiable by the UPPAAL model checker. The approach is implemented in the tool SARTS (http://sarts.boegholm.dk/).

However, in order for the schedulability analysis to be reliable and efficient, safe and tight estimates of the Worst-case execution time (WCET) of tasks must be provided. We show how real-time model checking and static analysis may be used to obtain safe and tight WCETs for programs running on platforms featuring caching and pipelining. The method works by constructing a UPPAAL model of the program being analysed and annotating the model with information from an inter-procedural value analysis. The program model is then combined with a model of the hardware platform and model checked for the WCET. Currently support for the platforms ARM7, ARM9 and ATMEL AVR 8-bit is available. The approach is implemented in the tool METAMOC (http://metamoc.martintoft.dk/).

The L4.verified Project and its next steps
Extended abstract

June Andronick

NICTA*, Australia

School of Computer Science and Engineering, UNSW, Sydney, Australia

june.andronick@nicta.com.au

The work presented here aims to tackle the general challenge of building truly trustworthy systems. This requires starting at the operating system (OS) level, and the most critical part of the OS is its kernel. The kernel is defined as the software that executes in the privileged mode of the hardware, meaning that there can be no protection from faults occurring in the kernel, and every single bug can potentially cause arbitrary damage. The concept of *microkernels* follow the idea [4] of minimizing the system's *trusted computing base* (TCB)—the part of the system that can bypass security. A microkernel, as opposed to the more traditional *monolithic* design of contemporary mainstream OS kernels, is reduced to just the bare minimum of code wrapping hardware mechanisms and needing to run in privileged mode. All OS services are then implemented as normal programs, running entirely in (unprivileged) user mode, and therefore can potentially be excluded from the TCB. A well-designed high-performance microkernel, such as the various representatives of the L4 microkernel family, consists of the order of 10 000 lines of code, making the trustworthiness problem more tractable.

The L4.verified project produced, in August 2009, the world's first formally proven correct general-purpose microkernel: seL4 [2]. As a microkernel, seL4 provides a minimal number of services to applications: abstractions for virtual address spaces, threads, inter-process communication (IPC). One of seL4's key differentiators is its fine-grained access control. The formal verification, from a high-level model down to low-level C code, was done using interactive, machine-assisted and machine-checked proof. Specifically, we used the theorem prover Isabelle/HOL [3]. Formally, our correctness statement is classic refinement: all possible behaviours of the C implementation are a subset of the behaviours of the abstract specification. The main assumptions of the proof are correctness of the C compiler and linker, assembly code, hardware, and boot code. The verification target was the ARM11 uniprocessor version of seL4, with an unverified x86 port. If the assumptions of the verification hold, we have mathematical proof that, among other properties, the seL4 kernel is free of buffer overflows, NULL pointer dereferences, memory leaks, and undefined execution. Another key benefit of a functional correctness proof is that proofs about the C implementation of

* NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program

the kernel can now be reduced to proofs about the specification for properties preserved by refinement.

The L4.verified project has demonstrated that with modern techniques and careful design, an OS microkernel is entirely within the realm of full formal verification. The next big step in the challenge of building truly trustworthy systems is to provide a framework to develop secure systems on top of seL4. Formally verifying programs with sizes approaching 10 000 lines of code is a significant improvement in what formal methods was previously able to verify with reasonable effort. However, 10 000 lines of code is still a significant limit on the application of formal methods to the verification of contemporary software systems. Modern software systems, beyond very simple embedded systems, frequently consist of millions of lines of code.

Our vision again follows the idea of minimizing the TCB and comes from the observation [1] that not all software in a large system necessarily contributes to a given property of interest, such as isolation or secure communication. Our approach is to develop methodologies and tools that enable developers to systematically (*i*) isolate the software parts that are not critical to a targeted property, and prove that nothing more needs to be verified about them for the specific property; and (*ii*) formally prove that the remaining critical parts satisfy the targeted property. This vision builds on, and is enabled by, the formal verification of the seL4 microkernel. The access control mechanism enforced by seL4 is used to isolate the identified large untrusted components, in a way that prevents them from violating a defined security property, leaving only the trusted components to be formally verified. The first steps of this approach have been demonstrated on a concrete example system, namely a multilevel secure access device aiming to isolate networked services of different classification levels. The system's security architecture has been designed to minimize the TCB to a single trusted component (in addition to the underlying kernel) and formalized in Isabelle/HOL. This formal security architecture has been used to prove that information cannot flow from one back-end network to another.

This case study illustrates our vision of how large software systems consisting of millions of lines of code can still have formal guarantees about certain targeted properties. This is achieved by building upon the access control guarantees provided by the verified seL4 microkernel and using it to isolate components such that their implementation need not be reasoned about.

References

1. J. Alves-Foss, P. W. Oman, C. Taylor, and S. Harrison. The MILS architecture for high-assurance embedded systems. *Int. J. Emb. Syst.*, 2:239–247, 2006.
2. G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *22nd SOSP*, pages 207–220, Big Sky, MT, USA, Oct 2009. ACM.
3. T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

4. J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proc. IEEE*, 63:1278–1308, 1975.

Clousot: Static contract checking with Abstract Interpretation

Francesco Logozzo[1]

Microsoft Research, Redmond, WA (USA)
logozzo@microsoft.com

A limiting factor to the adoption of formal methods in the everyday programming practice is that tools do not integrate well in the existing programming workflow. The price programmers has to pay to enjoy the benefits of formal methods include the use non-mainstream languages or non-standard compilers.

The CodeContracts project [2] at Microsoft aims at bridging the gap between formal specification and verification and a principle of least interference in the existing programmer's workflow. The main insight of CodeContracts is that specifications can be authored as code [1]. Contracts take form of method calls to a standard library. Therefore CodeContracts enable the programmer to write down specifications as Boolean expressions in their favorite .NET language (C#, F#, VB ...). This has several advantages: semantics of contracts is given by the IL produced by the compiler, no compiler modification is required, contracts are serialized and persisted as code (no need for separate parsing), all the IDE support (e.g. intellisense) the programmer is used to is automatically leveraged.

CodeContracts provide a standard and uniform way to describe contracts which can then be consumed by several tools. At Microsoft Research, we have developed a tool to automatically generate the documentation (ccdoc), to perform runtime checking (ccrewrite) and to perform static checking (cccheck/Clousot).

The static contract checker is based on abstract interpretation. It analyzes every method in isolation. The precondition of the method is turned into an assumption and the postcondition into an assertion. For public methods, the object invariant is assumed at the method entry and asserted at the exit point. For each method call, its precondition is asserted, and the postcondition assumed. The first phase of the analysis process resolves the heap (under some optimistic hypotheses e.g. on parameter aliasing), providing a scalar view of the program. On the top of that several value analyses are run to discover facts (including loop invariants) on the program. Value analyses include a non-null analysis, a numerical analysis [3], a pointer usage analysis and a container analysis. The value analyses propagate the initial (abstract) state through the method body performing a fixpoint computation with widening. The inferred facts are used to discharge proof obligations. Proof obligations are either explicit assertions in the code or semantic-induced ones such as non-null dereferences or array indexing. If a proof obligation cannot be discharged then the analysis is refined. One refinement is the use of a more expressive (yet expensive) abstract domain. If even this fails, then a backward analysis is performed to have a precise yet on demand handling of disjunctions. If the user turns on the opportune switch, then

Clousot uses the inferred information to extract the method postcondition and then to push it to all the callers.

CodeContracts can be downloaded at:
`http://research.microsoft.com/en-us/projects/contracts/`

References

1. Manuel Fähndrich, Michael Barnett, and Francesco Logozzo. Embedded contract languages. In *SAC '10: Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 2103–2110, New York, NY, USA, 2010. ACM.
2. Manuel Fähndrich, Mike Barnett, and Francesco Logozzo. Code Contracts, March 2009.
3. Vincent Laviron and Francesco Logozzo. Subpolyhedra: A (more) scalable approach to infer linear inequalities. In *VMCAI '09: Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 229–244, Berlin, Heidelberg, 2009. Springer-Verlag.

Adapting Components Using Interface Automata Enriched by the Action Semantics

Samir Chouali, Sebti Mouelhi, and Hassan Mountassir

Computer Sciences Laboratory (LIFC)
University of Franche-Comté
Besançon, FRANCE
{samir.chouali,sebti.mouelhi,hassan.mountassir}
@lifc.univ-fcomte.fr

Abstract. Reusability is one of the principal purposes of Component-Based Software Engineering (CBSE) and represents an important characteristic of a high-quality software component. It allows the use of components in diverse situations without affecting their codes. In this context, it is necessary to propose approaches to adapt a component with its environment when mismatches occur during their interactions. In this paper, we present a formal approach based on interface automata strengthened by the semantics of actions to adapt components. The elimination of mismatches is made at the signature, the protocol, and the semantic levels to ensure more flexible interoperability between components. Interface automata is a formalism intended to schedule between required and offered services of a component. In this work, the model is enriched by pre and post-conditions on parameters of component actions[1].

Keywords: Reusability, Action semantics, Interface automata, Component adaptation

1 Introduction

Component-based systems are made up of collection of interacting entities called components. The outline of component-based software engineering [19, 9] is to develop software applications not from scratch but by assembling various library components. This development approach allows software reuse without changing components codes. Consequently, one saves on development costs and time and one can extend component-based systems via plug and play components.

A component is a unit of composition with contractually specified interfaces and explicit dependencies [19]. An interface describes the services offered or required by a component without disclosing the component implementation. It is the only access to the information of a component. Interfaces may describe components at the level of action signatures (method signatures), behaviors or

[1] This work has received support from the The French National Research Agency, ANR-06-SETI-017 (TACOS).

protocols (scheduling of method calls), action semantics (method semantics), and the quality of services [12, 20].

Usually interoperability is not guaranteed during the component assembly and reuse. This is due to possible interface mismatches that may occur, between components, at the different levels cited above [6]. The reason is that components do not match perfectly the requirements of their environment. In this case, component adaptation should be performed in order to eliminate the resulting interface mismatches and ensure a more flexible interoperability between components. Software adaptation is the use of software entities, called *adaptors*, capable of enabling a correct interaction between components when mismatches occur at the level of signatures, protocols, and semantics.

Interfaces that expose protocol information of components can be specified naturally in an automaton-based language like *interface automata* [1, 2, 3]. It has the ability to model along a temporal order, both the input requirements (input actions) and the output behavior (output actions) of a component. The composition of two interfaces is achieved by synchronizing their shared output and input actions. An interesting verification approach was also proposed to detect interface incompatibilities that may occur when, from some states in the synchronized product, one automaton issues a shared action as output which is not accepted as input in the other. We say that those states are *illegal*. The proposed compatibility check approach of interface automata is *optimistic* [1]. Two interface automata A_1 and A_2 are compatible if there is an environment preventing their synchronization to enter illegal states. The composition approach of the other automata-based formalisms describing the interface protocols of components are considered pessimistic.

In this paper, we focus on adapting components whose behaviors are described by interface automata enriched by the semantics of action parameters. Actions are annotated by pre and post-conditions on parameters. In [8], we had treated only adaptation at the protocol level. Our purpose was to generate automatically an adaptor (interface automaton in-the-middle) for exactly two component interface automata according to a mapping that establishes a number of rules relating their mismatched input and output actions. The semantic adaptability between mismatched actions is checked before generating the adaptor by verifying the satisfiability of some implications between pre and post-conditions of the parameters of the mismatched actions. The automatic generation of the adaptor takes into account the orderings of actions of both interfaces and constructs progressively an automaton by consuming output actions before issuing outputs for their correspondent inputs in the mapping. One of the aims of this paper and the work in [8] is to bring together the optimistic composition approach of interface automata and adapting them at the signature, the protocol and the semantic levels. Therefore, our proposed adaptation presents a reliable way to interoperate components by verifying the semantics of their actions besides the elimination of mismatches, its principal role.

The paper is organized as follows. In section 2, we present our extended formalism based on interface automata approach to verify component interoper-

ability. In section 3, we describe the minimal specification of the action mismatch between two components. In section 4, we check the semantic adaptability of the mismatched actions. In section 5, we specify the adaptation of component behaviors using interface automata. Related works to our approach, the conclusion, and future works are presented in section 6 and 7.

2 Interface Automata Enriched by the Semantics of Action Parameters

Interface automata (IAs) have been defined by L. Alfaro and T. Henzinger [1], to model the temporal behavior of software component interfaces. These models are non-input-enabled I/O automata [13] which means that at every state some input actions may be non-enabled. Every component interface is described by one interface automaton where input actions are used to model methods that can be called, and the end of receiving messages from communication channels, as well as the return values from such calls. Output actions are used to model method calls, message transmissions via communication channels, and exceptions that occur during the methods executions. Local operations are called hidden actions. The alphabet of an interface automaton is built on the actions names of a component. This means that for each input action (or provided service) a in the component signature, there is an element $a?$ in the alphabet and for each output action (or required service) b, there is an element $b!$. A hidden action h is represented by the element $h;$ in the alphabet.

Definition 1 (Interface Automata). *An* interface automaton $A = \langle\ S_A,\ I_A,\ \Sigma_A^{\mathrm{I}},\ \Sigma_A^{\mathrm{O}},\ \Sigma_A^{\mathrm{H}},\ \delta_A\ \rangle$ *consists of*

- *a finite set* S_A *of states;*
- *a subset of initial states* $I_A \subseteq S_A$*. Its cardinality* $|I_A|^2 \leq 1$ *and* A *is called* empty *if* $I_A = \emptyset$*;*
- *three disjoint sets* $\Sigma_A^{\mathrm{I}}, \Sigma_A^{\mathrm{O}}$ *and* Σ_A^{H} *of input, output, and hidden actions names;*
- *a set* $\delta_A \subseteq S_A \times \Sigma_A \times S_A$ *of transitions between states.*

The input and output actions of an automaton A are called external actions uniformly ($\Sigma_A^{\mathrm{ext}} = \Sigma_A^{\mathrm{I}} \cup \Sigma_A^{\mathrm{O}}$) while output actions and internal actions are called locally-controlled actions ($\Sigma_A^{\mathrm{loc}} = \Sigma_A^{\mathrm{O}} \cup \Sigma_A^{\mathrm{H}}$). The set of hidden actions Σ_A^{H} may contain a special action *epsilon* ϵ that symbolizes the no-operation event. The set Σ_A refers to the set of all actions of A. We define by $\Sigma_A^{\mathrm{I}}(s)$, $\Sigma_A^{\mathrm{O}}(s)$, $\Sigma_A^{\mathrm{H}}(s)$, $\Sigma_A^{\mathrm{ext}}(s)$, $\Sigma_A^{\mathrm{loc}}(s)$, the input, output, internal, external, and local actions enabled at the state s. $\Sigma_A(s)$ denotes the set of all actions enabled from s.

Our approach extends interface automata by considering the action semantic to ensure a reliable verification of component interoperability. In [1], the check

[2] $|S|$ is the cardinality of some set S.

of the component compatibility is made only at the level of signatures and protocols, which are not sufficient to decide if two interfaces are compatible or not. Our contribution uses pre and post-conditions over the list of input and output parameters of components actions. These constraints on actions establish the semantic level of components interoperability.

The signature of an action a is the action name and the list of its parameters. An action may have n input parameters (where $n \geq 0$) belonging to the set P_a^{i} and at most one output parameter belonging to P_a^{o}. In the case where an action a has parameters, its signature is represented by $a(i_1,...,i_n) \rightarrow (o)$ where $i_1,...,i_n \in P_a^{\mathrm{i}}$ and $o \in P_a^{\mathrm{o}}$. The absence of parameters (input or output) is denoted by (). The semantics of external actions is depicted by pre and post-conditions defined over the list of parameters. Pre and post-conditions are propositional formulae of the propositional logic[3] built up from atomic assertions.

Components are black boxes whereof actions store some data and provide some externally visible behavior. Thus, the pre and post-conditions of action parameters have to be well-stated in such way no information about the action implementation are revealed. For example, given an action that computes the power of a number n and output the result in a parameter p. A legal post-condition of the action is "$p \geq 0$". The post-condition "$p = n \times n$" is not allowed. This is why we assume that, for the rest of the paper, arithmetic operators like addition and multiplication are not allowed in the action semantics.

Definition 2 (Action Semantics). *The* semantics Ψ_a *of an external action* a *is defined by the tuple* $\langle Pre_{\Psi_a}, Post_{\Psi_a} \rangle$ *where*

- Pre_{Ψ_a} *is defined in terms of input parameters. It is set to true if* a *has no input parameters* ($P_a^{\mathrm{i}} = \emptyset$)*;*
- $Post_{\Psi_a}$ *is defined in terms of both input and output parameters. It is set to true if* a *has no parameters* ($P_a^{\mathrm{i}} = P_a^{\mathrm{o}} = \emptyset$).

For a parameter p, we define a domain D_p which is a set of values that p can take. Atomic assertions used in the prepositions have the form p_1 * p_2 or p * v where p_1, p_2, and p are parameters of a given action, * $\in \{=,\neq\}$ and v is a valuation for the parameter p in D_p. For real and integer parameters, the operator * is in $\{=,\neq,<,>,\leq,\geq\}$.

Given an interface automaton A, we assume that for each transition (s,a,s') $\in \delta_A$ where a is external, the valuations of all parameters $p \in P_a^{\mathrm{i}} \cup P_a^{\mathrm{o}}$ have to be defined in D_p and they must not be in contradiction with the precondition Pre_{Ψ_a} and the post-condition $Post_{\Psi_a}$. We denote by Ψ_a^A, the semantics of the action a in Σ_A^{ext}.

The composition of two interface automata may take effect only if their actions are disjoint, except for shared input and output ones. Shared input and output actions must have the same number, the same type, the same order of input and output parameters. The parameter names of two shared actions may be different. During the composition of two interface automata, shared actions synchronize and all other actions are interleaved asynchronously.

[3] The operators of the propositional logic are $\{\wedge, \vee, \Leftarrow, \neg, \equiv\}$

Definition 3 (Composability). *Two interface automata A_1 and A_2 are* composable *if*

$$\Sigma^{\mathrm{I}}_{A_1} \cap \Sigma^{\mathrm{I}}_{A_2} = \Sigma^{\mathrm{O}}_{A_1} \cap \Sigma^{\mathrm{O}}_{A_2} = \Sigma^{\mathrm{H}}_{A_1} \setminus \{\epsilon\} \cap \Sigma_{A_2} = \Sigma^{\mathrm{H}}_{A_2} \setminus \{\epsilon\} \cap \Sigma_{A_1} = \emptyset.$$

Shared$(A_1,A_2) = (\Sigma^{\mathrm{I}}_{A_1} \cap \Sigma^{\mathrm{O}}_{A_2}) \cup (\Sigma^{\mathrm{I}}_{A_2} \cap \Sigma^{\mathrm{O}}_{A_1})$ is the set of shared input and output actions of A_1 and A_2. Suppose that the signature of an action $a \in$ *Shared*(A_1,A_2) is given by $a(i_1,...,i_n) \rightarrow (o)$ in A_1 and by $a(i'_1,...,i'_n) \rightarrow (o')$ in A_2 then, $D_{i_k} \subseteq D_{i'_k}$ for $1 \leq k \leq n$ and $D_o \subseteq D_{o'}$ in the case where $a \in \Sigma^{\mathrm{O}}_{A_1}$ and $a \in \Sigma^{\mathrm{I}}_{A_2}$. Otherwise, $D_{i_k} \supseteq D_{i'_k}$ for $1 \leq k \leq n$ and $D_o \supseteq D_{o'}$. This property is called the *domain inclusion* of the parameters of shared actions.

If all of the assumptions cited above are satisfied for the shared actions signatures of two interface automata A_1 and A_2 to be composed, we have to perform the renaming of parameter names in their pre and post-conditions.

Definition 4 (Parameter Renaming). *Given two composable interface automata A_1, A_2 where Shared$(A_1,A_2) \neq \emptyset$ and an action a in Shared(A_1,A_2), the signature of a is given by $a(i_{11},...,i_{n1}) \rightarrow (o_1)$ in A_1 and $a(i_{12},...,i_{n2}) \rightarrow (o_2)$ in A_2. The* renaming *of the parameters names in the semantics $\Psi_a^{A_1}$ and $\Psi_a^{A_2}$ is defined by the substitution of i_{12} by $i_{11},...,i_{n2}$ by i_{n1}, and o_2 by o_1 in $Pre_{\Psi_a^{A_2}}$ and $Post_{\Psi_a^{A_2}}$ or the substitution of i_{11} by $i_{12},...,i_{n1}$ by i_{n2}, and o_1 by o_2 in $Pre_{\Psi_a^{A_1}}$ and $Post_{\Psi_a^{A_1}}$.*

We denote by $\Psi_a^{A_1/A_2}$ and $\Psi_a^{A_2/A_1}$, the semantics of a shared external action a after the parameter renaming respectively in A_1 and A_2. We can now define the synchronized product $A_1 \otimes A_2$ properly.

Definition 5 (Synchronized Product). *Let A_1 and A_2 be two composable interface automata. The product $A_1 \otimes A_2$ is defined by*

- $S_{A_1 \otimes A_2} = S_{A_1} \times S_{A_2}$ *and* $I_{A_1 \otimes A_2} = I_{A_1} \times I_{A_2}$;
- $\Sigma^{\mathrm{I}}_{A_1 \otimes A_2} = (\Sigma^{\mathrm{I}}_{A_1} \cup \Sigma^{\mathrm{I}}_{A_2}) \setminus$ *Shared*(A_1, A_2);
- $\Sigma^{\mathrm{O}}_{A_1 \otimes A_2} = (\Sigma^{\mathrm{O}}_{A_1} \cup \Sigma^{\mathrm{O}}_{A_2}) \setminus$ *Shared*(A_1, A_2);
- $\Sigma^{\mathrm{H}}_{A_1 \otimes A_2} = \Sigma^{\mathrm{H}}_{A_1} \cup \Sigma^{\mathrm{H}}_{A_2} \cup$ *Shared*(A_1, A_2);
- $((s_1, s_2), a, (s'_1, s'_2)) \in \delta_{A_1 \otimes A_2}$ *if*
  - $a \notin$ *Shared*$(A_1, A_2) \wedge (s_1, a, s'_1) \in \delta_{A_1} \wedge s_2 = s'_2$
  - $a \notin$ *Shared*$(A_1, A_2) \wedge (s_2, a, s'_2) \in \delta_{A_2} \wedge s_1 = s'_1$
  - $a \in$ *Shared*$(A_1, A_2) \wedge ((s_1, a, s'_1) \in \delta_{A_1} \wedge a \in \Sigma^{\mathrm{O}}_{A_1}) \wedge ((s_2, a, s'_2) \in \delta_{A_2} \wedge a \in \Sigma^{\mathrm{I}}_{A_2}) \wedge$ $Pre_{\Psi_a^{A_1/A_2}} \Rightarrow Pre_{\Psi_a^{A_2/A_1}} \wedge$ $Post_{\Psi_a^{A_2/A_1}} \Rightarrow Post_{\Psi_a^{A_1/A_2}}$
  - $a \in$ *Shared*$(A_1, A_2) \wedge ((s_1, a, s'_1) \in \delta_{A_1} \wedge a \in \Sigma^{\mathrm{I}}_{A_1}) \wedge ((s_2, a, s'_2) \in \delta_{A_2} \wedge a \in \Sigma^{\mathrm{O}}_{A_2}) \wedge$ $Pre_{\Psi_a^{A_2/A_1}} \Rightarrow Pre_{\Psi_a^{A_1/A_2}} \wedge$ $Post_{\Psi_a^{A_1/A_2}} \Rightarrow Post_{\Psi_a^{A_2/A_1}}$

The incompatibility between A_1 and A_2 is due to (i) the existence of some states (s_1,s_2) in the product $A_1 \otimes A_2$ where one of the two interface automata outputs a shared action a from the state s_1 which is not accepted as input from the state s_2 or vice versa, or (ii) from that states they synchronize on the action signatures but their semantics do not match according to Definition 5. These states are called *illegal states*.

Definition 6 (Illegal States). *Given two composable interface automata A_1 and A_2, the set of* illegal states *Illegal*$(A_1,A_2) \subseteq S_{A_1} \times S_{A_2}$ *in* $A_1 \otimes A_2$ *is defined by* $\{(s_1, s_2) \in S_{A_1} \times S_{A_2} \mid \exists\ a \in \mathit{Shared}(A_1, A_2).$ *the following condition holds*$\}$

$$\left(\begin{array}{c}(a \in \Sigma^{\mathrm{O}}_{A_1}(s_1) \wedge a \notin \Sigma^{\mathrm{I}}_{A_2}(s_2)) \vee (a \in \Sigma^{\mathrm{O}}_{A_1}(s_1) \wedge a \in \Sigma^{\mathrm{I}}_{A_2}(s_2)) \\ \wedge (\mathit{Pre}_{\Psi_a^{A_1/A_2}} \not\Rightarrow \mathit{Pre}_{\Psi_a^{A_2/A_1}} \vee \mathit{Post}_{\Psi_a^{A_2/A_1}} \not\Rightarrow \mathit{Post}_{\Psi_a^{A_1/A_2}}))\end{array}\right)$$

$$\bigvee$$

$$\left(\begin{array}{c}(a \in \Sigma^{\mathrm{O}}_{A_2}(s_2) \wedge a \notin \Sigma^{\mathrm{I}}_{A_1}(s_1)) \vee (a \in \Sigma^{\mathrm{O}}_{A_2}(s_2) \wedge a \in \Sigma^{\mathrm{I}}_{A_1}(s_1)) \\ \wedge (\mathit{Pre}_{\Psi_a^{A_2/A_1}} \not\Rightarrow \mathit{Pre}_{\Psi_a^{A_1/A_2}} \vee \mathit{Post}_{\Psi_a^{A_1/A_2}} \not\Rightarrow \mathit{Post}_{\Psi_a^{A_2/A_1}}))\end{array}\right)$$

The reachability of states in $\mathit{Illegal}(A_1, A_2)$ do not implies that A_1 and A_2 are not compatible. The existence of an environment E (an interface automaton) that produces appropriate inputs for the product $A_1 \otimes A_2$ ensures that illegal states will not be entered and then A_1 and A_2 can be used together. The compatible states, denoted by $\mathit{Comp}(A_1,A_2)$, are states from which the environment can prevent entering illegal states.

Definition 7 (Compatibility). *Given two composable interface automata A_1 and A_2, A_1 and A_2 are* compatible *if and only if the initial state of their product $A_1 \otimes A_2$ is compatible.*

The *composition* $A_1 \parallel A_2$ of two compatible interface automata A_1 and A_2 is defined by (i) $S_{A_1\|A_2} = \mathit{Comp}(A_1,A_2)$, (ii) $I_{A_1\|A_2} = I_{A_1\otimes A_2} \cap \mathit{Comp}(A_1,A_2)$, (iii) $\Sigma^*_{A_1\|A_2} = \Sigma^*_{A_1\otimes A_2}$ where $* \in \{\mathrm{O}, \mathrm{I}, \mathrm{H}\}$, and (iv) $\delta_{A_1\|A_2} = \delta_{A_1\otimes A_2} \cap \mathit{Comp}(A_1,A_2) \times \Sigma_{A_1\|A_2} \times \mathit{Comp}(A_1,A_2)$.

The verification steps in this approach are the same as those detailed in [1] except that we consider the semantics of actions. The proposed algorithm of the compatibility decides if two interface automata are compatible by checking if their composition is non-empty. Our extentions do not increase the linear complexity of the previous proposed one. Finally, we add that the associative criterion of the composition operator $\parallel$ holds between three interface automata and it is undefined when some of them are not composable.

3 Interface Mismatches

The definitions of component interface mismatches [7, 5, 6] are essentially due to the reuse of components in a system design which is often harmed by mismatch cases such as: (i) names of exchanged messages between components do not correspond which may lead to deadlock situations, components regularly interact on the same action names; (ii) the orderings of messages or actions in both component protocols do not correspond; (iii) an action in a component that has no counterpart in the other one, or correspond to more than one action.

For interface automata, the behavioural mismatch cannot be detected by applying the synchronized product between two composable interface automata as it was defined in Definition 4, because the case where there is no correspondence

between the action names leads to them being absent from the set of shared actions. Thus, all of mismatched actions are interleaved asynchronously in the product. To avoid this constraint, our adaptation specification starts by establishing an abstract way to denote the composition requirements. We corroborate the explicit description of interactions between components thanks to *rules*. They relate the mismatched actions used in different components which are supposed to implement some interactions. Rules relate actions even if they do not really label some transitions in the automaton as required by the optimistic approach of interface automata.

The minimal adaptor specification of two interface automata A_1 and A_2 is the set of rules called a *mapping*. The mapping does not represent any behavioural detail about the adaptor.

Definition 8 (Rules and Mappings). *A* rule α *for two composable interface automata* A_1 *and* A_2*, is a couple* $\langle L_1, L_2 \rangle \in (2^{\Sigma^{O}_{A_1}} \times 2^{\Sigma^{I}_{A_2}}) \cup (2^{\Sigma^{I}_{A_1}} \times 2^{\Sigma^{O}_{A_2}})$[4] *such that* $(L_1 \cup L_2) \cap Shared(A_1, A_2) = \emptyset$ *and if* $|L_1| > 1$ *(or* $|L_2| > 1$*) then* $|L_2| = 1$ *(or* $|L_1| = 1$*);*

A mapping $\Phi(A_1, A_2)$ *for two composable interface automata* A_1 *and* A_2 *is a set of rules* α_i*, for* $1 \leq i \leq |\Phi(A_1, A_2)|$.

According to Definition 8, a rule in our approach deals with one-to-one, many-to-one, and one-to-many correspondences between actions. More clearly, the adaptation may in general relate either an action or a group of actions of one automaton with one action in the other. For instance, a client authenticates itself by sending first its user name and then a password while the server accepts both data in a single login shot. We denote the set of the mismatched actions by $Mismatch_{\Phi}(A_1, A_2) = \{a \in \Sigma^{ext}_{A_1} \cup \Sigma^{ext}_{A_2} \mid \exists\ \alpha \in \Phi(A_1, A_2)\ .\ a \in \Pi_1(\alpha) \vee a \in \Pi_2(\alpha)\}$[5].

Given two composable interface automata A_1 and A_2 and a mapping $\Phi(A_1, A_2)$, if $\Phi(A_1, A_2) = \emptyset$, the adaptation of A_1 and A_2 has no sense and their synchronization is defined by their product $A_1 \otimes A_2$ as it was defined in section 2. Otherwise, we proceed on two steps:

- we check first the semantic adaptability between the mismatched actions in the mapping $\Phi(A_1, A_2)$;
- if the semantic adaptability check was successfully made without giving rise to incompatibilities, we generate the adaptor of A_1 and A_2 according to the mapping $\Phi(A_1, A_2)$. If the generated adaptor is non-empty and it is compatible with both of A_1 and A_2, we say that A_1 and A_2 are *adaptable*.

4 Semantic Adaptability

The semantic adaptability between the mismatched actions of two composable interface automata has to be made before generating the adaptor. The mis-

[4] For some set S, 2^S is its power set.

[5] $\Pi_1(\langle a, b \rangle) = a$ and $\Pi_2(\langle a, b \rangle) = b$ are respectively the projection on the first element and the second element of the couple $\langle a, b \rangle$.

matched actions have to respect some constraints at the level of their semantics. Let us consider two interface automata A_1 and A_2 and a given mapping $\Phi(A_1, A_2)$. To perform the semantic adaptability check between A_1 and A_2 according to $\Phi(A_1, A_2)$, it is required that for each rule $\alpha = \langle L_1, L_2 \rangle \in \Phi(A_1, A_2)$ the following conditions hold:

1. $\sum_{a \in L_1} \mid P_a^{\mathrm{i}} \mid = \sum_{b \in L_2} \mid P_b^{\mathrm{i}} \mid$;
2. $\sum_{a \in L_1} \mid P_a^{\mathrm{o}} \mid = \sum_{b \in L_2} \mid P_b^{\mathrm{o}} \mid$;
3. if $|L_1| = 1$ and $\mid L_2 \mid \geq 1$ where $L_1 = \{a\}$, $L_2 = \{b_1, ..., b_{|L_2|}\}$, and $P_a^{\mathrm{o}} = \{o_a\}$ then there exists exactly one action $b_k \in L_2$ ($1 \leq k \leq \mid L_2 \mid$) such that $P_{b_k}^{\mathrm{o}} = \{o_{b_k}\}$, $P_{b_l}^{\mathrm{o}} = \emptyset$ for $1 \leq l \leq \mid L_2 \mid$ and $l \neq k$, and the two output parameters o_a and o_{b_k} have to satisfy the domain inclusion condition:
   - if $L_1 \subseteq \Sigma_{A_1}^{\mathrm{O}}$, then $D_{o_a} \subseteq D_{o_{b_k}}$;
   - else $D_{o_a} \supseteq D_{o_{b_k}}$;

   θ_α denotes the tuple (a,b_k). If $P_a^{\mathrm{o}} = \{\}$, (a,b_k) is not defined;
4. the condition is analogous to the previous one with $|L_1| \geq 1$ and $|L_2| = 1$ where $L_1 = \{a_1, ..., a_{|L_1|}\}$ and $L_2 = \{b\}$;
5. there exists a function $\varphi_\alpha^{\mathrm{i}} : \bigcup_{a \in L_1} P_a^{\mathrm{i}} \rightarrow \bigcup_{b \in L_2} P_b^{\mathrm{i}}$ that associates each input parameter p of actions in L_1 with an input parameter q of actions in L_2. The function $\varphi_\alpha^{\mathrm{i}}$ have to satisfy the domain inclusion condition:
   - if $L_1 \subseteq \Sigma_{A_1}^{\mathrm{O}}$, then $D_p \subseteq D_{\varphi_\alpha^{\mathrm{i}}(p)}$ where $p \in \bigcup_{a \in L_1} P_a^{\mathrm{i}}$;
   - else $D_{\varphi_\alpha^{\mathrm{i}}(p)} \subseteq D_p$ where $p \in \bigcup_{a \in L_1} P_a^{\mathrm{i}}$.

The first and the second conditions state that the number of input (respectively output) parameters of actions in L_1 is equal to the number of input (respectively output) parameters of actions in L_2. The third condition states the relations between the output parameter of the action $a \in L_1$ and the one of the action $b_k \in L_2$. We assume that the other actions in $L_2 \setminus \{b_k\}$ have no output parameters. The intuition behind these conditions is to avoid conflicts between the pre and post-conditions during the semantic adaptability check by ensuring the equality between the number of input and output parameters.

The renaming of the input and output parameter in the semantics of actions in $\mathit{Mismatch}_\Phi(A_1, A_2)$ is defined as follows.

- For all $a \in L_1$ and $b \in L_2$, it is defined by the substitution of each input parameter i of a in $\mathit{Pre}_{\Psi_a^{A_1}}$ and $\mathit{Post}_{\Psi_a^{A_1}}$ by $\varphi_\alpha^{\mathrm{i}}(i)$ or the substitution of each input parameter i' of b in $\mathit{Pre}_{\Psi_b^{A_2}}$ and $\mathit{Post}_{\Psi_b^{A_2}}$ by ${\varphi_\alpha^{\mathrm{i}}}^{-1}(i')$[6] if $\varphi_\alpha^{\mathrm{i}}$ is defined.
- If the tuple $\theta_\alpha = (a,b)$ exists, it is defined by the substitution of the output parameter o_a in $\mathit{Post}_{\Psi_a^{A_1}}$ by o_b or the substitution of the output parameter o_b in $\mathit{Post}_{\Psi_b^{A_2}}$ by o_a.

We denote by $\Psi_a^{A_1,\alpha}$ and $\Psi_b^{A_2,\alpha}$ respectively the semantics of actions a in $\Pi_1(\alpha)$ and actions b in $\Pi_2(\alpha)$ after the parameter renaming.

[6] For a function f, we define by f^{-1} its inverse function.

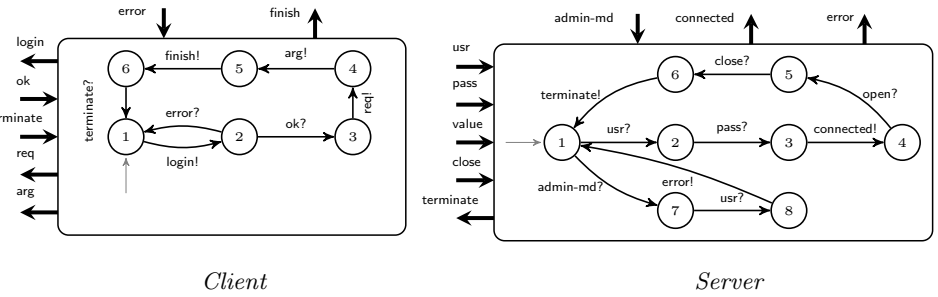


Fig. 1. A variant of a client/server system

Definition 9 (Semantic Adaptability). *Given two interface automata A_1 and A_2 and a non empty mapping $\Phi(A_1, A_2)$, the* semantic adaptability *of a rule α in $\Phi(A_1, A_2)$ is defined by the following conditions:*

1. *if $\Pi_1(\alpha) \subseteq \Sigma^{\mathrm{O}}_{A_1}$, then*

$$\left(\begin{array}{c} \bigwedge\limits_{a \in \Pi_1(\alpha)} Pre_{\Psi_a^{A_1,\alpha}} \Rightarrow \bigwedge\limits_{b \in \Pi_2(\alpha)} Pre_{\Psi_b^{A_2,\alpha}} \\ \wedge \\ \bigwedge\limits_{a \in \Pi_1(\alpha)} Post_{\Psi_a^{A_1,\alpha}} \Leftarrow \bigwedge\limits_{b \in \Pi_2(\alpha)} Post_{\Psi_b^{A_2,\alpha}} \end{array} \right)$$

2. *if $\Pi_1(\alpha) \subseteq \Sigma^{\mathrm{I}}_{A_1}$, then the condition is analogous to the first one by inversing the implications.*

We say that A_1 and A_2 are semantically adaptable *according to the mapping $\Phi(A_1, A_2)$ if the semantic adaptability of each rule $\alpha \in \Phi(A_1, A_2)$ holds.*

The semantic adaptability conditions are stated in a similar way as the semantic compatibility of the shared actions defined in Definition 5 except that for adaptation, we treat sets of mismatched actions associated by the rules of the mapping.

Example 1. Let us consider the two composable interface automata *Client* and *Server* shown in Figure 1 and a mapping Φ(*Client*,*Server*) = {⟨ {*login*}, {*usr*, *pass*}⟩, ⟨{*finish*}, {*close*}⟩, ⟨{*ok*}, {*connected*}⟩ ⟨{*req*,*arg*}, {*open*}⟩ }. The set *Shared*(A_1, A_2) = {*error*,*terminate*}.

After authentication, *Client* sends a request *req*! to open a file in read-only or write mode. After that, it sends an action *arg*! containing the name of a file to be open. *Server* receives the two actions by executing an action *open*? that open the file in readonly or write mode. After using the file, *Client* sends a signal *finish*! indicating to *Server* that the file is ready to be closed (action *close*?). Finally, *Server* sends a signal *terminate*! to terminate the session. The action *admin-md*? is a super signal received from the administrator of the system to

Table 1. The signatures of actions in $Mismatch_{\Phi}$(*Client*,*Server*)

	Client	***Server***
α_1	login(uname,passwd,lu,lp)→(exist)	usr(username,lengthu)→()
		pass(password,lengthp)→(exist)
α_2	ok(msg)→()	connected(logmsg)→()
α_3	req(read)→()	open(readonly,filename)→(open)
	arg(file)→(status)	
α_4	finish()→(status)	close()→(closed)

Table 2. The semantics of actions in $Mismatch_{\Phi}$(*Client*,*Server*)

Client	***Server***
$Pre_{\Psi^{Client}_{login}} \equiv 1 < \text{lu} \leq 20 \wedge 8 \leq \text{lp} \leq 10$ $Post_{\Psi^{Client}_{login}} \equiv \text{exist} = 1 \vee \text{exist} = 0$	$Pre_{\Psi^{Server}_{usr}} \equiv 1 < \text{lengthu} \leq 30$ $Post_{\Psi^{Server}_{usr}} \equiv \text{true}$
	$Pre_{\Psi^{Server}_{pass}} \equiv 6 \leq \text{lengthp} \leq 10$ $Post_{\Psi^{Server}_{pass}} \equiv \text{exist} = 1 \vee \text{exist} = 0$
$Pre_{\Psi^{Client}_{ok}} \equiv \text{true}$ $Post_{\Psi^{Client}_{ok}} \equiv \text{true}$	$Pre_{\Psi^{Server}_{connected}} \equiv \text{true}$ $Post_{\Psi^{Server}_{connected}} \equiv \text{true}$
$Pre_{\Psi^{Client}_{req}} \equiv \text{read} = 0 \vee \text{read} = 1$ $Post_{\Psi^{Client}_{req}} \equiv \text{true}$	$Pre_{\Psi^{Server}_{open}} \equiv \text{readonly} = 0 \vee \text{readonly} = 1$ $Post_{\Psi^{Server}_{open}} \equiv \text{open} = 1 \vee \text{open} = 1$
$Pre_{\Psi^{Client}_{arg}} \equiv \text{true}$ $Post_{\Psi^{Client}_{arg}} \equiv \text{status} = 0 \vee \text{status} = 1$	
$Pre_{\Psi^{Client}_{finish}} \equiv \text{true}$ $Post_{\Psi^{Client}_{finish}} \equiv \text{status} = 0 \vee \text{status} = 1$	$Pre_{\Psi^{Server}_{close}} \equiv \text{true}$ $Post_{\Psi^{Server}_{close}} \equiv \text{closed} = 0 \vee \text{closed} = 1$

open a super user session. When a client username is received by the server after receiving the *admin-md*! signal from an administrator, then an error is detected.

The mismatched actions are described and classified by the rules in Table 1. The function $\varphi^{\text{i}}_{\alpha_2}$ is defined by {*msg*↦*logmsg*}. The function $\varphi^{\text{i}}_{\alpha_4}$ is not defined. The function $\varphi^{i}_{\alpha_1}$ is defined by and {*uname*↦*username*, *lu*↦*lengthu*, *passwd*↦*password*, *lp*↦*lengthp*}. The function $\varphi^{\text{i}}_{\alpha_3}$ is defined by {*read*↦*readonly*, *file*↦*filename*}. The function $\varphi^{\text{i}}_{\alpha_4}$ is empty. $\theta_{\alpha_1} = (login,pass)$, θ_{α_2} is not defined, $\theta_{\alpha_3} = (arg,open)$, and $\theta_{\alpha_4} = (finish,close)$. The parameters *uname*, *passwd*, *username*, *password*, *msg*, *logmsg*, *file*, and *filename* are strings. The parameters *lu*, *lp*, *lengthu*, *lengthp*, *read*, *readonly*, *status*, *open*, and *closed* are integers. As the reader can conclude, the conditions to perform the semantic adaptability check hold for all α in $\Phi(A_1, A_2)$:

– for all $\alpha \in \Phi(A_1, A_2)$, $\sum_{a \in \Pi_1(\alpha)} |P^{\text{i}}_a| = \sum_{b \in \Pi_2(\alpha)} |P^{\text{i}}_b|$ and $\sum_{a \in \Pi_1(\alpha)} |P^{\text{o}}_a| = \sum_{b \in \Pi_2(\alpha)} |P^{o}_b|$;

– the domain inclusion conditions are satisfied for θ_* and φ_*^i where $* \in \Phi(\mathit{Client}, \mathit{Server})$.

The semantics of the mismatched actions respectively for *Client* and *Server* are listed in Table 2. After unifying the mismatched actions in $\mathit{Mismatch}_\Phi(\mathit{Client}, \mathit{Server})$, the reader can easily verify the semantic adaptability for all α in $\Phi(\mathit{Client}, \mathit{Server})$ holds. For example, for the rule α_1, $\mathit{Pre}_{\Psi_{login}^{Client,\alpha_1}} \Rightarrow (\mathit{Pre}_{\Psi_{usr}^{Server,\alpha_1}} \wedge \mathit{Pre}_{\Psi_{pass}^{Server,\alpha_1}})$ is satisfiable $((1 < \mathrm{lu} \leq 20 \wedge 8 \leq \mathrm{lp} \leq 10) \Rightarrow (1 < \mathrm{lu} \leq 30 \wedge 6 \leq \mathrm{lp} \leq 10))$. Also, $\mathit{Post}_{\Psi_{login}^{Client,\alpha_1}} \Leftarrow (\mathit{Post}_{\Psi_{usr}^{Server,\alpha_1}} \wedge \mathit{Post}_{\Psi_{pass}^{Server,\alpha_1}})$ is satisfiable $((\mathrm{exist} = 1 \vee \mathrm{exist} = 0) \Leftarrow (\mathrm{true} \wedge (\mathrm{exist} = 1 \vee \mathrm{exist} = 0)))$. We can deduce that *Client* and *Server* are semantically adaptable according to $\Phi(\mathit{Client}, \mathit{Server})$.

5 Adaptor Specification and Construction

After verifying the semantic adaptability between two composable interface automata A_1 and A_2 according to a mapping $\Phi(A_1, A_2)$, we treat in this section the IA specification and construction of their adaptor. The adaptor must be composable with A_1 and A_2 and satisfy the event reordering of both A_1 and A_2.

Given an interface automaton A, we denote by $\Theta_A^{\mathrm{S}}(s) \subseteq S_A^*$ the set of successor finite runs $\theta = s_1 a_1 s_2 a_2 ... s_n$ such that $s_1 = s$, s_n is the initial state or a state that has no outgoing transitions, and for all $1 \leq i < n$, there is a transition $(s_i, a_i, s_{i+1}) \in \delta_A$. We denote by $\Theta_A^{\mathrm{P}}(s) \subseteq S_A^*$ the set of predecessor finite runs $\theta = s_1 a_1 s_2 a_2 ... s_n$ is defined exactly as $\Theta_A^{\mathrm{S}}(s)$ except $s_1 = i$ where $i \in I_A$ and $s_n = s$. The set Θ_A of all finite runs of A equals to $\Theta_A^{\mathrm{S}}(i)$ where $i \in I_A$. We say that a succession of transitions $s_1 a_1 s_2 a_2 ... s_n$ (for $n \geq 2$) is included in a run σ in $\Theta_A^{\mathrm{S}}(s)$ or $\Theta_A^{\mathrm{P}}(s)$ (denoted by the operator $\sqsubseteq$), if all transitions of $s_1 a_1 s_2 a_2 ... s_n$ are transitions of σ.

Definition 10 (Adaptor). *Given two composable interface automata A_1, A_2, and a mapping $\Phi(A_1, A_2)$, an* adaptor *for A_1 and A_2 according to the mapping $\Phi(A_1, A_2)$ is an interface automaton $Ad = \langle\ S_{Ad},\ I_{Ad},\ \Sigma_{Ad}^{\mathrm{I}},\ \Sigma_{Ad}^{\mathrm{O}},\ \Sigma_{Ad}^{\mathrm{H}},\ \delta_{Ad}\ \rangle$ such that*

– $\Sigma_{Ad}^{\mathrm{I}} = \{a \mid a \in \mathit{Mismatch}_\Phi(A_1, A_2) \cap (\Sigma_{A_1}^{\mathrm{O}} \cup \Sigma_{A_2}^{\mathrm{O}})\}$;
  . *for all $a \in \Sigma_{Ad}^{\mathrm{I}}$, $\Psi_a^{Ad} = \Psi_a^{A_1}$ if $a \in \Sigma_{A_1}^{\mathrm{O}}$. Otherwise, $\Psi_a^{Ad} = \Psi_a^{A_2}$;*
– $\Sigma_{Ad}^{\mathrm{O}} = \{a \mid a \in \mathit{Mismatch}_\Phi(A_1, A_2) \cap (\Sigma_{A_1}^{\mathrm{I}} \cup \Sigma_{A_2}^{\mathrm{I}})\}$;
  . *for all $a \in \Sigma_{Ad}^{\mathrm{O}}$, $\Psi_a^{Ad} = \Psi_a^{A_1}$ if $a \in \Sigma_{A_1}^{\mathrm{I}}$. Otherwise, $\Psi_a^{Ad} = \Psi_a^{A_2}$;*
– $\Sigma_{Ad}^{\mathrm{H}} \subseteq \{\epsilon\}$;
– $\delta_{Ad} \subseteq S_{Ad} \times \Sigma_{Ad}^{\mathrm{I}} \cup \Sigma_{Ad}^{\mathrm{O}} \cup \{\epsilon\} \times S_{Ad}$;
– $\mathit{Shared}(Ad, A_1) = \bigcup_{\alpha \in \Phi(A_1, A_2)} \Pi_1(\alpha)$;
– $\mathit{Shared}(Ad, A_2) = \bigcup_{\alpha \in \Phi(A_1, A_2)} \Pi_2(\alpha)$;
– *For all $s \in S_{Ad}$ and $\sigma \in \Theta_{Ad}^{\mathrm{P}}(s)$, if there exist $r_1 a_1 ... r_n a_n s \sqsubseteq \sigma$ and $\alpha \in \Phi(A_1, A_2)$ such that $\Pi_i(\alpha) \subseteq \Sigma_{A_i}^{\mathrm{O}}$ for $i \in \{1, 2\}$ and $\Pi_i(\alpha) \subseteq \bigcup_{k \in 1..n} \{a_k\}$, then for all $\rho \in \Theta_{Ad}^{\mathrm{S}}(s)$, there exists $s b_1 ... b_m t_m \sqsubseteq \rho$ such that $\Pi_{3-i}(\alpha) \subseteq \Sigma_{A_{3-i}}^{\mathrm{I}}$ and $\Pi_{3-i}(\alpha) \subseteq \bigcup_{l \in 1..m} \{b_l\}$.*

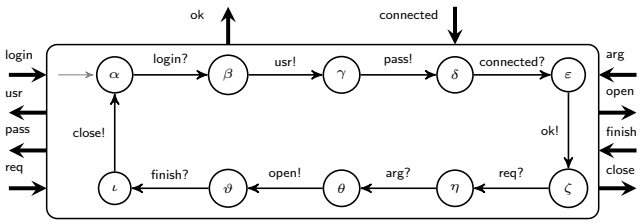


Fig. 2. The adaptor *Adaptor* for *Client* and *Server*

The last condition of Definition 10 states that for all $\sigma \in \Theta_A$, if $\exists \alpha \in \Phi(A_1, A_2)$ such that the output actions (enabled as input in Ad) of α are present in σ then they are succeed by there correspondent input actions (enabled as output in Ad).

Property 1. An adaptor Ad for two interface automata A_1 and A_2 according to a mapping $\Phi(A_1, A_2)$ is composable with A_1 and A_2.

The property can be easily verified according to Definition 10. The composition of A_1 and A_2 is performed by synchronizing first Ad with either A_1 or A_2, computing their composition according to our extended approach, and then by composing the resulting composition with the remaining automaton. We suppose that the actions of the adaptor have the same signatures and semantics as actions in $Mismatch_{\Phi}(A_1,A_2)$. If the composite interface automaton $A_1 \parallel Ad \parallel A_2$ is non empty then A_1 and A_2 are compatible after their adaptation at the protocol and the semantic levels.

Our presented algorithm presented in [8] constructs an adaptor for two composable interface automata A_1, A_2, and a given non empty mapping $\Phi(A_1, A_2)$. The algorithm is basically a loop which reads in parallel A_1 and A_2 and constructs as one goes along the set of states and the set of transitions of the adaptor. The algorithm is executed by respecting the reordering of events of both interfaces A_1 and A_2. The algorithm marks and removes from the generated graph all the fragments of runs that do not respect the last condition of Definition 10. If the result of the algorithm is non-empty then we check that the generated adaptor Ad is compatible with both of A_1 and A_2. In that case, we say that the two A_1 and A_2 are adaptable. If $A_1 \parallel Ad \parallel A_2$ is non empty we say that A_1 and A_2 are compatible after their adaptation by Ad.

The part of the algorithm that constructs the set of states and transitions has the time complexity $\mathbf{O}(|S_{A_1} \times S_{A_2}|.(|\delta_{A_1}| + |\delta_{A_2}|))$. The time complexity of the part that removes the undesired run fragments is linear in the number of states of the generated states.

Example 2. As the reader can conclude, *Adaptor* is composable with both *Client* and *Server* presented in Example 1 and it satisfies all the items of Definition 10. Our proposed algorithm in [8] generates exactly the same interface automaton

shown in Figure 2. Suppose that the semantic compatibility between the shared actions *error* and *terminate* holds, then *Adaptor* is compatible with both *Client* and *Server*. The composite interface automaton (*Client* $\|$ *Adaptor*) $\|$ *Server* is non empty which makes *Client* and *Server* compatible after their adaptation.

6 Related Works

Several techniques of adaptation show how to automatically derive adaptors in order to eliminate mismatches between components during their interactions. In [21], the authors propose an interesting approach based on finite state machines to adapt components specified by interfaces describing component protocol and action signatures. This approach deals with one-to-one relations between actions. In [14], the authors propose the Smart Connectors approach which allows the construction of adaptors using the provided and required interfaces of the components in order to resolve partial matching problems in COTS component acquisition.

In [5], the authors have proposed a formal approach based on calculus to generate automatically adaptors using the Prolog language. In [10], Hemer has proposed, using template from the CARE language, to define adaptation strategies for modifying and combing components. In [15], the authors have proposed a model of adaptors expressed in the B formal method, allowing to define the interoperability between components. In [17] the authors introduce the concept of parameterized contracts and a model for component interfaces, they also present algorithms and tools for specifying and analyzing component interfaces in order to check interoperability and to generate adapted component interfaces. Finally, Bosch [4] gives a large overview on adaptation mechanisms including non-automated approaches can be found in [11, 18].

The approaches described above propose solutions for the component adaptation based on different specification formalisms of component interfaces. Our approach is different from the others, because we propose a solution to adapt particular components that are specified by interface automata. This formalism allows to exploit optimistic approach [1] to check to component interoperability. This adaptation approach deals with the signature, the semantic, and the protocol levels, and deals also with possibly complex adaptation scenarios : one-to-one and one-to-many correspondences between actions.

7 Conclusion and Future Works

In this paper, we propose a formal approach for the automatic development of component adaptors, allowing the elimination of mismatches between interacting components. Our component interfaces are described by interface automata enriched by the action semantics. We propose to describe in interface automata component information at three levels: signature, semantic(signature and semantic of offered and required actions), and component protocol(interactive behavior that the component follows). We specify a correspondence mapping between the

mismatched actions of two components as a first abstract specification of the adaptor. This mapping deals with one-to-one and one-to-many correspondences between the actions. A compatibility check after adaptation is made on the set of mismatched action in a similar way as the set of shared actions. We propose an algorithm that generates automatically the adaptor for two composable interface automata according to a fixed mapping. The generated adaptor allows to eliminate mismatches at the signature, semantic and the protocol levels. The proposed algorithms were implemented in Java in order to validate them, and we plan to propose a complete tool in the future works.

We are developing a tool that implements a framework checking the compatibility between interface automata at the protocol and the semantic levels [16]. We plan also to implement the proposed adaptation approach in our framework.

References

[1] L. Alfaro and T. A. Henzinger. Interface automata. *ACM Press, 9th Annual Aymposium of FSE (Foundations of Software Engineering)*, pages 109–120, 2001.

[2] L. Alfaro and T. A. Henzinger. Interface theories of component-based design. *In the proceeding of the First International Workshop of Embedded Software (EMSOFT), LNCS*, 2211:148–165, 2001.

[3] L. Alfaro and T. A. Henzinger. Interface-based design. *NATO Science Series : Mathematics, Physics, and Chemistry, Engineering Theories of Software Intensive Systems*, 195:83–104, 2005.

[4] J. Bosch. Design and use of software architectures - adopting and evolving a product-line approach. *Addison-Wesley, Reading, MA, USA*, 2000.

[5] A. Bracciali, A. Brogi, and C. Canal. A formal approach to component adaptation. *Journal of Systems and Software*, 74:45–54, 2005.

[6] C. Canal, J. Murillo, and P. Poizat. Software adaptation. *Special Issue on Software adaptation*, 12(1):9–31, 2006.

[7] C. Canal, P. Poizat, and G. Salaün. Synchronizing behavioural mismatch in software composition. *Proc. of FMOODS'06, LNCS*, 6:63–77, 2006.

[8] S. Chouali, S. Mouelhi, and H. Mountassir. Adapting component behaviours using interface automata. *IEEE Computer Society proceedings, Euromicro SEAA 2010 conference*, September 2010.

[9] G. T. Heineman and H. M. Ohlenbusch. An evaluation of component adaptation techniques. In *The International Workshop on Component-Based Software Engineering*, 1999.

[10] D. Hemer. A formal approach to component adaptation and composition. *In Proceedings of the Twenty-eighth Australasian conference on Computer Science ACSC '05 Newcastle, Australia*, pages 259–266, 2005.

[11] S. Kent, C. Ho-Stuart, , and P. Roe. Negotiable interfaces for components. *Journal of Object Technology, TOOLS USA proceedings 1*, pages 249–265, 2002.

[12] D. Konstantas. Interoperation of object oriented application. *In Proceedings of Object-Oriented Software Composition, Oscar Nierstrasz and Dennis Tsichritzis, Prentice Hall*, pages 69–95, 1995.

[13] N. Lynch and M. Tuttle. Hierarcical correctness proofs for distributed algorithms. *In the proceeding of the 6th ACM Symp. Principles of Distributed Computing*, pages 137–151, 1987.

[14] H. Min, S. Choi, and S. Kim. Using smart connectors to resolve partial matching problems in cots component acquisition. *LNCS, Springer-Verlag, Berlin, Germany*, 3054:40–47, 2004.

[15] I. Mouakher, A. Lanoix, and J. Souquières. Component Adaptation: Specification and Verification. In *11th International Workshop on Component Oriented Programming (WCOP 2006)*, page 8, ECOOP 2006, Nantes, France, 07 2006.

[16] S. Mouelhi, S. Chouali, and H. Mountassir. Refinement of interface automata strengthened by action semantics. *ENTCS, FESCA09 of the European joint conference on Theory and Practice of Software (ETAPS'09)*, 253-1:111–126, March 2009.

[17] R. Reussner. Automatic component protocol adaptation with the coconut/j tool suite. *Future Generation Computer Systems*, 19(5):627–639, 2003.

[18] H. Schmidt and R. Reussner. Generating adaptors for concurrent component protocol synchronisation. *In the proceeding of the Fifth IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems*, pages 213–229, 2002.

[19] C. Szyperski. Component software: Beyond object oriented programming. *Addison Wesley*, 1999.

[20] P. Wegner. Interoperability. *ACM Computing Survey*, 28:285–287, 1996.

[21] D. Yellin and R. Strom. Protocol specifications and components adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333, 1997.

CVPP: A Tool Set for Compositional Verification of Control–Flow Safety Properties

Marieke Huisman[1] and Dilian Gurov[2]*

[1] University of Twente, Netherlands
[2] Royal Institute of Technology, Stockholm, Sweden

Abstract. This paper describes CVPP, a tool set for compositional verification of control–flow safety properties for programs with procedures. The compositional verification principle that underlies CVPP is based on maximal models constructed from component specifications. Maximal models replace the actual components when verifying the whole program, either for the purposes of modularity of verification or due to unavailability of the component implementations at verification time. A characteristic feature of the principle and the tool set is the distinction between program structure and behaviour. While behavioural properties are more abstract and convenient for specification purposes, structural ones are easier to manipulate, in particular when it comes to verification or the construction of maximal models. Therefore, CVPP also contains the means to characterise a given behavioural formula by a set of structural formulae. The paper presents the underlying framework for compositional verification and the components of the tool set. Several verification scenarios are described, as well as wrapper tools that support the automatic execution of such scenarios, providing appropriate pre– and post–processing to interface smoothly with the user and to encapsulate the inner workings of the tool set.

1 Introduction

To enable verification of realistic software, verification techniques have to be compositional and algorithmically decidable. Compositionality ensures that the verification task can be split up in smaller pieces, while algorithmic decidability ensures that verification can be done automatically, without any user interaction. Moreover, for many application domains, compositionality and algorithmic decidability are essential.

For example, in a dynamically reconfigurable distributed system, components can join and leave the system at run–time dynamically. For such an *open system*, appropriate verification techniques are necessary to support safe downloading, *i.e.*, to determine without any user interaction whether a newly arriving component will not corrupt the well–functioning of the global system. These techniques require the *relativisation* of the correctness of the system on the specifications

* Partially funded by the EU FET project FP7–ICT–2009–3 HATS.

and the local correctness of its components. This relativisation can also be used for the purposes of *modularity*. Modular verification is a means of controlling the complexity of verifying large software. It allows an independent local evolution of the implementations of individual modules without affecting the global correctness of the program.

The CVPP tool set is designed to tackle exactly this kind of verification problems by supporting an algorithmic technique for compositional verification. Its focus is on control–flow safety properties of programs with (possibly recursive) procedures. Such properties typically describe sets of allowed sequences of method invocations, and are conveniently expressed in temporal logic. The underlying program model is that of *flow graphs*, abstracting completely from program data to allow efficient algorithmic modular verification. However, the model can be enhanced with exception information or multi–threading. Even though the tool set is developed with compositionality in mind, it can also be used for non–compositional control–flow verification problems of programs with procedures. In particular, it allows to reduce infinite–state verification of behavioural properties to finite–state verification of structural properties.

Abstracting away from all data may seem like a severe restriction, but still many useful properties can be expressed, such as:

- there are no calls to non–atomic methods within atomic transactions;
- in a voting system, candidate selection has to be finished, before the vote can be confirmed;
- a method that changes sensitive data is only called from within a dedicated authentication method, i.e., unauthorized access is not possible;
- in a door access control system, the password has to be checked before the door is unlocked, and it can only be changed if the door is unlocked.

Extending the technique with data over finite domains will allow for a wider range of properties and possible applications, but needs to be combined with abstraction techniques to control the complexity of verification. Such an extension will be investigated in future work.

The present paper describes CVPP, its underlying compositional verification framework, and its implementation. We describe three important verification scenarios: (*i*) open system verification, (*ii*) modular verification, and (*iii*) non–compositional verification. We also discuss the encapsulation of the inner workings of CVPP by means of wrapper tools that automate the various scenarios.

Previous work by the authors on tool support and case studies has been reported in 2004 [15]. The current version of the tool set, discussed in this paper, includes later extensions: (*i*) an inliner to abstract private methods [10], (*ii*) more general program models concerning exceptions, threads and open flow graphs [14,12], and (*iii*) a property translation from behavioural to structural properties [11,12]. The last extension allowed local assumptions to be behavioural, whereas before they had to be structural. Further, we have unified the inputs and outputs to allow interoperability of the individual tools, and have started to work on wrapper tools, automating the verification scenarios.

Related Work Maven is a modular verification tool addressing temporal properties of procedural languages in the context of aspects [8]. A non–compositional verification method based on a program model closely related to ours is presented by Alur and others [3]. It proposes a temporal logic CaRet for nested calls and returns (generalised to a logic for nested words in [1]) that can be used to specify regular properties of local paths within a procedure that skips over calls to other procedures.

Most of the existing work on modular verification of safety properties is based on Hoare logic. Müller was the first to propose a sound modular Hoare–style verification technique for object–oriented languages [17]. A typical verification tool within this line of work is Spec# [4].

Recent work by Alur and Chauhuri proposes a unification of Hoare–style and Manna–Pnueli–style temporal reasoning for procedural programs, presenting proof rules for procedure–modular temporal reasoning [2].

Organisation Sections 2 and 3 sketch the tool set's theoretical background and underlying verification method. Section 4 describes the different tools that make up CVPP, followed by a description of typical verification scenarios in Section 5. Section 6 exemplifies some typical verification tasks when using CVPP. We conclude with possible extensions that would make CVPP applicable to a larger class of problems (without changing the underlying methodology).

2 Program Model and Logic

This section summarises the program model and logic that underlies CVPP. For a more detailed account, the reader is referred to [13].

As mentioned earlier, a characteristic feature of CVPP is the distinction between structural and behavioural properties. Usually, we are interested in properties of the behaviour of a program, while its structure is just a means for accomplishing the desired behaviour. Furthermore, the same behaviour can be produced by several structures. It is thus more natural and more abstract to specify programs with behavioural properties than with structural ones.

However, algorithmic techniques for program analysis and verification are computationally considerably more expensive on the level of program behaviour than on the level of program structure. Program correctness problems are therefore often phrased in terms of the program structure rather than in terms of its behaviour. Furthermore, many behavioural properties have natural structural counterparts, *e.g.*, tail recursion, while other behavioural properties can be characterised through finite sets of structural ones (see Section 3). Therefore, CVPP is set up in such a way that structural properties can be used whenever this is possible and meaningful.

2.1 Model and Logic

Our program model is control–flow based and thus over–approximates actual program behaviour. It defines two different views on programs: a structural and

a behavioural one. Both views are instantiations of the general notions of model, defined below. Notice in particular that these instantiations yield a structural and a behavioural version of the logic, and that this enables a uniform treatment of structure and behaviour whenever possible.

Definition 1. (Model) *A* model *is a structure* $\mathcal{M} = (S, L, \rightarrow, A, \lambda)$*, where* S *is a set of states,* L *a set of labels,* $\rightarrow \subseteq S \times L \times S$ *a labelled transition relation,* A *a set of atomic propositions,* $\lambda\colon S \rightarrow \mathcal{P}(A)$ *a valuation, assigning to each state* s *the set of atomic propositions that hold in* s*. An* initialised model *is a pair* $(\mathcal{M}, E)$*, with* $\mathcal{M}$ *a model and* $E \subseteq S$ *a set of entry states.*

As property specification language we use the fragment of the modal μ-calculus [16] with boxes and greatest fixed-points only. This temporal logic is capable of characterising simulation (*cf.* [13]) and is thus suitable for expressing safety properties. Throughout, we fix a set of labels L, a set of atomic propositions A, and a set of propositional variables V.

Definition 2. (Logic) *The formulae of our logic are inductively defined by:*

$$\phi ::= p \mid \neg p \mid X \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid [a]\,\phi \mid \nu X.\phi$$

where $p \in A$*,* $a \in L$ *and* $X \in V$*.*

Satisfaction on states $(\mathcal{M}, s) \models \phi$ is defined in the standard fashion [16]. For instance, formula $[a]\,\phi$ holds of state s in model $\mathcal{M}$ if ϕ holds in all states accessible from s via an edge labelled a. A model $(\mathcal{M}, E)$ satisfies a formula ϕ, denoted $(\mathcal{M}, E) \models \phi$, if all its entry states E satisfy ϕ. The constant formulae *true* (denoted tt) and *false* (ff) are definable. For convenience, we use $p \Rightarrow \phi$ to abbreviate $\neg p \vee \phi$. We assume that formulae have pair–wise distinct fixed–point binders, and unless stated otherwise, are closed and guarded (*cf.* [22]).

2.2 Control–Flow Structure and Behaviour

Control–Flow Structure We abstract away from all data, therefore program structure is defined as a collection of control–flow graphs (or flow graphs), one for each of the program's methods. Let *Meth* be a countably infinite set of method names. A method graph is an instance of the general notion of model.

Definition 3. (Method graph) *A* method graph *for* $m \in Meth$ *over a finite set* $M \subseteq Meth$ *of method names is an initialised model* $(\mathcal{M}_m, E_m)$*, where* $\mathcal{M}_m = (V_m, L_m, \rightarrow_m, A_m, \lambda_m)$ *is a finite model and* $E_m \subseteq V_m$ *a non–empty set of* entry points *of* m*.* V_m *is the set of* control nodes *of* m*,* $L_m = M \cup \{\varepsilon\}$*,* $A_m = \{m, r\}$*, and* $\lambda_m\colon V_m \rightarrow \mathcal{P}(A_m)$ *is defined so that* $m \in \lambda_m(v)$ *for all* $v \in V_m$ (i.e., *each node is tagged with its method name). The nodes* $v \in V_m$ *with* $r \in \lambda_m(v)$ *are* return points.

Example 1. Figure 1 shows a simple Java class and the (simplified) flow graph it induces. The flow graph consists of two method graphs - one for method `even` and one for method `odd`. Entry nodes are depicted as edges without source.

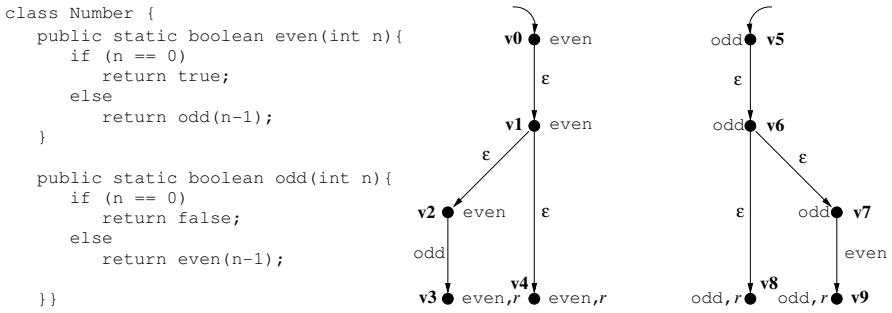


Fig. 1. A simple Java class and its flow graph

Flow graph *interfaces* are defined as pairs $I = (I^+, I^-)$, where $I^+, I^- \subseteq \mathit{Meth}$ are finite sets of names of *provided* and (externally) *required* methods, respectively[3]. A flow graph $\mathcal{G}$ with interface I is denoted $\mathcal{G} : I$. The flow graph of a program is essentially the (disjoint) union $\uplus$ of its method graphs. Flow graphs can only be composed if their interfaces match. A flow graph is *closed* if $I^- = \varnothing$, *i.e.,* it does not require any external methods. Satisfaction, instantiated to flow graphs, is called structural satisfaction $\models_s$.

Example 2. Consider the flow graph in Example 1. The property "on every path from a program entry node, the first encountered call edge goes to a return node" is formalised by the structural formula $\nu X.\,[\mathtt{even}]\,r \wedge [\mathtt{odd}]\,r \wedge [\varepsilon]\,X$, in effect specifying that the program is tail–recursive.

Control–Flow Behaviour Next, we instantiate models on the behavioural level. Transition label τ designates internal transfer of control, $m_1\ \mathsf{call}\ m_2$ designates an invocation of method m_2 by method m_1, and $m_2\ \mathsf{ret}\ m_1$ designates the corresponding return.

Definition 4. (Behaviour) *Let $\mathcal{G} = (\mathcal{M}, E) : I$ be a closed flow graph where $\mathcal{M} = (V, L, \to, A, \lambda)$. The* behaviour *of $\mathcal{G}$ is defined as the initialised model $b(\mathcal{G}) = (\mathcal{M}_b, E_b)$, where $\mathcal{M}_b = (S_b, L_b, \to_b, A_b, \lambda_b)$, such that $S_b = V \times V^*$,* i.e., *states are pairs of control points v and stacks σ (also called* configurations*), $L_b = \{m_1\ k\ m_2 \mid k \in \{\mathsf{call}, \mathsf{ret}\}, m_1, m_2 \in I^+\} \cup \{\tau\}$, $A_b = A$, $\lambda_b((v,\sigma)) = \lambda(v)$, and $\to_b \subseteq S_b \times L_b \times S_b$ is defined by the rules:*

$$
\begin{array}{rll}
[\text{transfer}] & (v,\sigma) \xrightarrow{\tau}_b (v',\sigma) & \text{if } m \in I^+,\ v \xrightarrow{\varepsilon}_m v',\ v \models \neg r \\
[\text{call}] & (v_1,\sigma) \xrightarrow{m_1\ \mathsf{call}\ m_2}_b (v_2, v_1' \cdot \sigma) & \text{if } m_1, m_2 \in I^+,\ v_1 \xrightarrow{m_2}_{m_1} v_1', \\
 & & \quad v_1 \models \neg r,\ v_2 \models m_2,\ v_2 \in E \\
[\text{return}] & (v_2, v_1 \cdot \sigma) \xrightarrow{m_2\ \mathsf{ret}\ m_1}_b (v_1,\sigma) & \text{if } m_1, m_2 \in I^+,\ v_2 \models m_2 \wedge r,\ v_1 \models m_1
\end{array}
$$

[3] We only require I^- to contain methods that are not provided by I^+. This is different from our earlier work (*e.g.,* [13]), but in line with the tool set implementation.

The set of initial configurations is defined by $E_b = E \times \{\epsilon\}$*, where* ϵ *denotes the empty sequence over* V*.*

The definition is easily extended to open flow graphs (see [12]). Flow graph behaviour can alternatively be defined via *pushdown automata* (PDA) [13, Def. 34] and approximated with the related notion of pushdown systems (PDS). We exploit this by using PDS model checking for verification of behavioural properties (see [6]). Currently, our tool set relies on the external tool Moped [19]; however, this requires the properties to be translated in LTL.

Example 3. Consider the flow graph from Example 1. Because of possible unbounded recursion, it induces an infinite–state behaviour. One example execution of the program is represented by the following path (in the branching structure) from an initial to a final configuration:

$$(v_0, \epsilon) \xrightarrow{\tau}_b (v_1, \epsilon) \xrightarrow{\tau}_b (v_2, \epsilon) \xrightarrow{\text{even call odd}}_b (v_5, v_3) \xrightarrow{\tau}_b (v_6, v_3) \xrightarrow{\tau}_b$$
$$(v_7, v_3) \xrightarrow{\text{odd call even}}_b (v_0, v_9 \cdot v_3) \xrightarrow{\tau}_b (v_1, v_9 \cdot v_3) \xrightarrow{\tau}_b$$
$$(v_4, v_9 \cdot v_3) \xrightarrow{\text{even ret odd}}_b (v_9, v_3) \xrightarrow{\text{odd ret even}}_b (v_3, \epsilon)$$

Also on the behavioural level, we instantiate the definition of satisfaction: we define $\mathcal{G} \models_b \phi$ as $b(\mathcal{G}) \models \phi$. The resulting behavioural logic is powerful enough to express the class of *security policies* defined by finite state security automata [18].

Example 4. For the flow graph from Example 1, the behavioural formula $\texttt{even} \Rightarrow \nu X.\, [\texttt{even call even}]\, \mathsf{ff} \wedge [\tau]\, X$ expresses the property "in every program execution starting in method `even`, the first call is not to method `even` itself".

Extensions This section presents the basic program model and logic, considering only normal, sequential control–flow. Extensions with exceptions and with multi–threaded behaviour (with synchronisation on locks) exist [14], and are supported in CVPP. The extension to open flow graphs mentioned above is also supported. In ongoing work we address further extensions to Boolean programs, as well as to richer fragments of the μ–calculus; this is not incorporated in CVPP yet.

3 Framework for Compositional Verification

The compositional verification method underlying our tool set is based on the computation of maximal models from component specifications and the instantiation of components with these models when model checking global system properties. For finite–state systems, this approach was introduced in [9] and since then it has become a standard technique for reducing the verification of correctness of property decompositions to model checking.

Maximal Models for Compositional Verification A model is said to be *maximal* for a given property ϕ, if it satisfies ϕ and simulates (*w.r.t.* a suitable property-preserving simulation relation $\leq$) all models satisfying ϕ. For models in the sense

of Definition 1 and formulae in the logic from Definition 2, maximal models exist and are unique up to isomorphism (see [13]). To compute a maximal model for a property ϕ, we present the formula as a modal equation system (see [5]), which is then transformed into a canonical form, the so–called *simulation normal form.* A formula ϕ in simulation normal form can be directly mapped into a (finite) model $\mathcal{M}$ that simulates all models that satisfy ϕ; *i.e.*, for any model $\mathcal{M}'$: $\mathcal{M}' \leq \mathcal{M} \Leftrightarrow \mathcal{M}' \models \phi$. Due to this close connection between simulation and satisfaction, we obtain the following sound and complete verification principle [13]:

> *Compositional verification principle for models*: to show $\mathcal{M}_1 \uplus \mathcal{M}_2 \models \psi$, it suffices to show $\mathcal{M}_1 \models \phi$ (*i.e.,* component $\mathcal{M}_1$ satisfies a suitably chosen *local assumption* ϕ) and $\mathcal{M}_\phi \uplus \mathcal{M}_2 \models \psi$ (*i.e.,* component $\mathcal{M}_2$, when composed with the maximal model $\mathcal{M}_\phi$ for ϕ, satisfies the *global guarantee* ψ).

Completeness of the principle implies that no *false negatives* exist: if $\mathcal{M}_\phi \uplus \mathcal{M}_2 \models \psi$ fails, then there is indeed a model $\mathcal{M}$ such that $\mathcal{M} \models \phi$ but $\mathcal{M} \uplus \mathcal{M}_2 \not\models \psi$.

Adaptation of this principle to flow graphs (as models) and structural and behavioural properties presents us with certain difficulties. Given a structural or behavioural flow graph property ϕ, there is no guarantee that the maximal model of ϕ is a legal flow graph structure or behaviour.

Maximal Flow Graphs from Structural Specifications For structural properties this problem can be solved for a given flow graph interface I, because we can characterise precisely the flow graphs having interface I as models through a structural formula θ_I in our logic. Let $I = \{m_1, m_2\}$ be a closed flow graph interface. A model is a flow graph with this interface exactly when it satisfies the formula $\theta_I = (\nu X.m_1 \wedge [m_1, m_2, \varepsilon]X) \vee (\nu Y.m_2 \wedge [m_1, m_2, \varepsilon]Y)$, which essentially expresses that edges in the flow graph do not cross method boundaries. Then, for every structural formula ϕ, the maximal model of the formula $\phi \wedge \theta_I$ is a flow graph $\mathcal{G}_{\phi,I}$ that simulates structurally all flow graphs with interface I that satisfy ϕ. We term this flow graph the *maximal flow graph* for formula ϕ and interface I, and the compositional verification principle formulated above still applies for flow graphs and structural properties. The above compositional verification principle can then be adapted to structural properties of flow graphs, yielding the following sound and complete compositional verification principle, presented as a proof rule (see [13] for technical details):

$$(\mathsf{struct-comp})\ \frac{\mathcal{G}_1 \models_s \phi \qquad \mathcal{G}_{\phi, I_{\mathcal{G}_1}} \uplus \mathcal{G}_2 \models_s \psi}{\mathcal{G}_1 \uplus \mathcal{G}_2 \models_s \psi}\ \mathcal{G}_1 : I_{\mathcal{G}_1}$$

Maximal Flow Graphs from Behavioural Specifications In the case of behavioural flow graph properties, however, there is no such way to characterise in our logic all models that constitute behaviours of flow graphs with a given interface (intuitively, this is because the logic is not capable of expressing context–free properties). Furthermore, these models are infinite–state and cannot be constructed

explicitly; what we actually need is a way to construct the maximal flow graph for a given behavioural formula ϕ and interface I. It turns out, however, that in general there is no such single flow graph, but rather a set of flow graphs having the property that every flow graph satisfying ϕ is simulated by some flow graph in the set. To compute such a set, we have developed a translation from behavioural flow graph properties ϕ to equivalent sets of structural properties $\Pi_I(\phi)$ for a given interface I. The translation is based on a tableau construction that conceptually amounts to symbolic execution of the behavioural formula, collecting structural constraints along the way. By keeping track of the subformulae that have been examined, recursion in the structural constraints is identified and captured by fixed–point formulae (for details see [11]). Combining this translation with maximal flow graph generation for structural properties yields the following sound and complete compositional verification principle for flow graphs and behavioural properties, presented as a proof rule:

$$(\mathsf{beh}-\mathsf{comp})\ \frac{\mathcal{G}_1 \models_b \phi \qquad \left\{\mathcal{G}_{\chi,I_{\mathcal{G}_1}} \uplus \mathcal{G}_2 \models_b \psi\right\}_{\chi\in\Pi_{I_{\mathcal{G}_1}}(\phi)}}{\mathcal{G}_1 \uplus \mathcal{G}_2 \models_b \psi}\ \mathcal{G}_1 : I_{\mathcal{G}_1}$$

In addition, we have also developed a "mixed" rule [13], where local structural assumptions are combined with global behavioural guarantees.

The presented proof rules are flexible, so that they allow reasoning about a combination of concrete components (*i.e.*, given through their implementation) and abstract components (*i.e.*, given though their specification), both at the structural and the behavioural levels. Section 5 shows typical verification scenarios, where these proof rules are applied for open system and modular verification. A possible instantiation of this approach is to choose individual methods as components. The proof rules then give rise to a procedure–modular verification technique for temporal properties, see [20].

4 Tool Support for Compositional Verification

This section describes the different internal data formats and tools within the CVPP tool set. It also exemplifies the different input formats used. A high–level overview of CVPP's architecture is shown in Figure 2 (where rounded boxes denote data formats, squared boxes tool components, and dashed lines denote external formats or tools).

As program input format, currently the Java bytecode format is used. Internally, there are three important *data formats:*

- *Model*: the program model representation, containing nodes, edges, a valuation and a set of entry points.
- *Formula*: the property representation. We support behavioural and structural formulae in our logic, both in recursive and in equation system form.

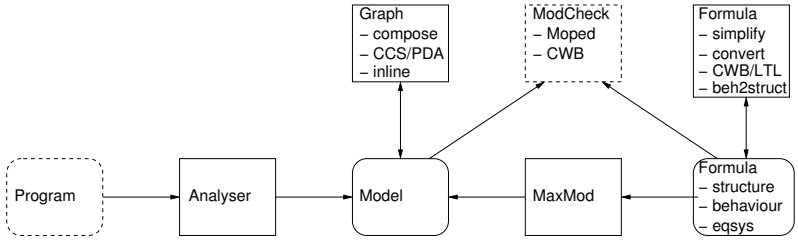


Fig. 2. The CVPP tool set architecture

– *Interface:* the interface representation, containing lists of provided and of externally required methods. Interfaces are used as auxiliary information by almost all tool components, and therefore we did not include it explicitly in Figure 2.

The *components* of the tool set are the following:

– Analyser: from Java classes to flow graphs. Java bytecode classes are abstracted into flow graphs. The tool is build on top of the Soot framework [21].
– Graph: transformations on the program model representations. The main operations supported are flow graph composition, pretty printing in different formats (in particular as CCS process terms and as PDS of the induced behaviour), and inlining of private methods. The use of the latter operation, called Graph Inliner, is briefly explained in Section 5.1 (see also [10]).
– Formula: transformations on the property representations. The main operations supported are the simplification of formulae, the conversion from one property format to another (such as the translation of our logic from recursive to equation system form, needed for maximal model construction), pretty printing as a CWB or LTL formula (as input for Moped), as well as the characterisation of behavioural formulae by structural ones. The latter operation is referred to as Beh2Struct. In addition, we allow properties to be expressed using so–called *patterns.* Patterns provide abbreviations for commonly used specification constructs. They increase readability and make the property more independent of the interface. The Formula component translates patterns into our logic.
– MaxMod: the maximal model construction as described in Section 3. This component uses formulae expressed as equation systems.
– ModCheck: model checking, using external tools: for structural properties we use CWB, the Edinburgh Concurrency Workbench [7], while for behavioural properties we rely on Moped, a PDS model checker for LTL [19].

To conclude this section, we show how the examples from Section 2 are written in CVPP's input formats. Consider again the flow graph from Figure 1. The method graph of method `even` is written as follows:

```
 interface for Number: provided even, odd
struct. formula Ex. 2: nu X.(([even] r) /\ ([odd] r) /\ ([eps] X))
  beh. formula Ex. 4: meth(even) => nu X.(([even call even] ff) /\ ([tau] X))
```

Fig. 3. Examples in CVPP's input format

```
node 0 meth(even) entry        edge 0 1 eps
node 1 meth(even)              edge 1 2 eps
node 2 meth(even)              edge 1 4 eps
node 3 meth(even) ret          edge 2 3 odd
node 4 meth(even) ret
```

Figure 3 exemplifies how interfaces and structural and behaviour properties are written in CVPP's input format.

5 Typical Verification Scenarios

Section 3 presented several compositional verification principles; this section describes in detail some typical scenarios supported by CVPP and these verification principles. In addition, we also describe how CVPP can be used for non–compositional verification. This is in particular interesting for behavioural properties: by means of the translation of behavioural properties into structural ones, CVPP provides an effective way to reduce the verification problem for behavioural properties to the computationally simpler problem for structural ones.

5.1 Open System Verification

The most general application of the proof rules presented in Section 3 is to *open system* verification, where some components are given by an implementation (referred to here as concrete components), while others are only given by a specification (abstract components). This can typically happen with dynamically reconfigurable or evolving software, where some components are either not known or simply not statically fixed at verification time.

Thus, verification of a global property of an open system has to be relativised on the local specifications of the abstract components. For instance, if all specifications are behavioural, this is achieved by consecutively applying rule (beh − comp) on every abstract component. The implementations of the abstract components, once available, are checked against their local specifications.

An additional complexity stems from the detail of information in the concrete components. Often these will contain information about private methods, that are not visible to other components. In contrast, the abstract components and global properties are typically described at the level of the public interface. Therefore, the implementation details in the concrete components are abstracted away, by using the Graph Inliner, to the publicly visible behaviour, before composing the components.

The overall verification task thus divides into two independent tasks, supported by our tool set as follows:

1. *Local correctness*: Check whether the implementation, once available, of every abstract component meets its local specification as described below in Section 5.3.
2. *Global correctness*:
   (a) for every concrete component, from its implementation, extract a flow graph using the Analyser, and use the Graph Inliner to construct its publicly visible behaviour;
   (b) for every abstract component, if its local specification is behavioural, translate the property to an equivalent set of structural ones using Beh-2Struct;
   (c) for every structural property, being either a local specification of an abstract component itself or resulting from step 2(b), compute a maximal flow graph using MaxMod;
   (d) for all instantiations of abstract components by corresponding constructed maximal flow graphs, and instantiations of concrete components by their extracted flow graphs, compose the graphs using Graph to produce a global flow graph of the system, and model check the latter against the global specification as described below in Section 5.3.

5.2 Modular Verification

In the modular software design paradigm the goal is to verify the modules of a software system locally, *i.e.*, independently of each other, and then to combine the local correctness arguments into a global correctness proof of the whole system. In our verification framework, modular verification is simply an instance of the more general case of open system verification described above, with modules as components and where all components are abstract. This eliminates task 2(a) and simplifies conceptually task 2(d).

One can view the notion of module on different levels of granularity. One (rather extreme) case in procedural programming languages is when every procedure itself is considered a module and is equipped with a specification. In this case we obtain *procedure–modular* verification, similar to many Hoare logic based verification approaches. We have recently shown on a case study that it is indeed possible and convenient to reason at this level of granularity about control–flow safety properties of an application [20].

5.3 Non–compositional Verification

The open system and modular verification scenarios above give rise to several non-modular verification tasks. In addition, CVPP also can be used to reason in a fully non-compositional setting. This is in particular useful to reason about behavioural properties. Due to unbounded recursion, verification of behavioural properties for procedural programs is infinite–state, even when all

data is abstracted away as in our program model. On the other hand, verification of structural properties is finite–state. Thus, by applying our translation from behavioural to sets of structural properties, one can reduce verification of behavioural properties to a finite number of finite–state verification tasks.

With our tool set, given a Java application and a property specification (either behavioural or structural), perform the following steps:

1. extract the flow graph of the application using the Analyser (and if necessary, use the Graph Inliner to abstract away from implementation details);
2. if the property is structural, cast the flow graph as a CCS term using Graph, and model check the term against the property using the CWB;
3. if the property is behavioural, there are two alternatives: either
   (a) cast the flow graph as a pushdown system using Graph, and model check it against the property using Moped; or
   (b) translate the property to an equivalent set of structural ones using Beh-2Struct, and perform step 2 for each one of these.

Step 3(b) is particularly meaningful in settings where the behavioural specifications are known in advance (such as the security policies of mobile platforms) and are relatively stable; the property translation can then be applied prior to the verification task itself.

5.4 Wrapper Tools for Standard Verification Scenarios

The different scenarios described above require the use of several of the tools of CVPP in a particular pre–defined order. Therefore, to make CVPP easier to use, and to hide away the internal formats and translations within the tool set, *wrapper tools* are being developed that perform the typical verification scenarios automatically. A wrapper implements a pre– and a post–processor that translates input and output of the tool set, and performs the different verification steps automatically. The post–processor appropriately handles feedback from the model checkers: when a structural property is violated, it is indicated where in the program this violation occurs; when a behavioural property is violated the model checking counter example is translated back into a program trace.

The first wrapper tool that we developed is ProMoVer [20]. This automates procedure–modular verification of Java programs annotated with global and method–local specifications. ProMoVer is evaluated on a small but realistic case study: we verified the absence of calls to non–atomic methods within Java Card transactions for a Java Card electronic purse application[4]. In the near future, we plan to develop wrapper tools for the other scenarios.

[4] A web–based interface to ProMoVer is available from:
`http://www.csc.kth.se/~siavashs/ProMoVer/promover.php`.

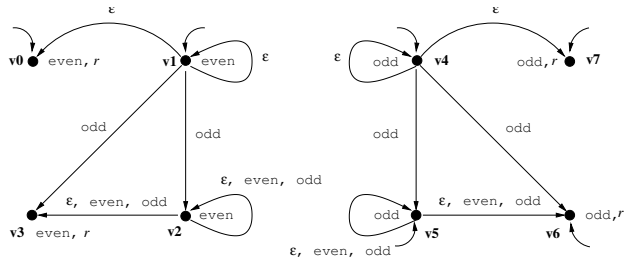


Fig. 4. Maximal flow graph for "the first call is not to method `even` itself"

6 Executing the Verification Scenarios

To illustrate how CVPP is used, this section discusses how parts of the different verification scenarios described in the previous section are applied on concrete examples. For a larger example discussing our experiences with ProMoVer for the verification of the safe use of the Java Card transaction mechanism in an e-commerce application for smart cards, we again refer the reader to [20].

6.1 Generating Maximal Flow Graphs for a Behavioural Property

One important subtask in the compositional verification scenarios discussed in the previous section is the construction of maximal flow graphs from a behavioural specification of a component; see steps 2(b,c) of the open system verification scenario. As explained in Section 3, this is achieved by translating the behavioural property into an equivalent set of structural ones, and by constructing a maximal flow graph for each of the latter.

For example, consider a component specified by an interface where methods `even` and `odd` are provided and no external methods are required, and by the behavioural property "in every program execution starting in method `even`, the first call is not to method `even` itself" formalised in Example 4. Providing this interface and formula to Beh2Struct, and optimising the result with the simplification facility of Formula, we obtain one structural formula: $\texttt{even} \Rightarrow \nu X.\,[\texttt{even}]\,\mathsf{ff} \wedge [\epsilon]\,X$. To compute a maximal flow graph, we first apply the conversion facilities of Formula to transform the formula into a modal equation system, which is then passed on, together with the original interface, to MaxMod. The resulting maximal flow graph is shown in Figure 4. Notice that the method graphs for `even` and `odd` are isomorphic, but the graph of method `even` has two entry nodes while the graph of method `odd` has four; as a result, the former restricts the behaviour in that, once called, method `even` can only call method `odd` as a first method call, while the latter makes no restrictions on the behaviour whatsoever. This maximal flow graph can now be substituted for the given component when model checking global system properties.

6.2 Closed System Model Checking of a Behavioural Property

Consider again the component of the previous subsection, described by the interface where methods `even` and `odd` are provided and no external methods are required, and by the behavioural property in Example 4. We want to show that the class `Number` defined in Example 1 is an appropriate implementation of this component. This is an instance of the non-compositional verification scenario in Section 5.3. Thus, using the Analyser, we first extract the flow graph, resulting in the flow graph as in Figure 1. For this application, there is no difference between public and private interface, thus there is no need to use the Graph Inliner.

The property is behavioural, thus we have a choice (*cf.* step 3, Section 5.3). *(a)* We can model check the behavioural property directly. We use Graph to produce the PDS from the flow graph, and Formula to transform the property to an LTL formula. Then Moped is used to verify that class `Number` indeed respects this property. *(b)* As in the previous subsection, we can compute the structural formula that characterises the behavioural formula by using Beh2Struct. We use Graph to pretty print the flow graph as CCS term and Formula to pretty print the formula in CWB's input format. Then CWB is used to verify that class `Number` indeed respects this structural property.

7 Conclusion

CVPP is a tool set for compositional verification of control–flow safety properties of procedural programs. It supports a completely automatic verification method based on maximal models. The underlying general compositional verification principle instantiates to two important verification scenarios, namely open system verification and modular verification. By means of an algorithmic translation of behavioural into structural properties, the tool is also applicable to non–compositional verification, allowing infinite–state PDA model checking to be reduced to standard finite–state model checking. The various scenarios can be supported by wrapper tools, such as ProMoVer, that encapsulate the inner workings of the tool set and provide a smooth interface to the user.

The largest CVPP case study so far is the verification of absence of illicit applet interactions in a smart card application [13,6]. This has been redone with the later extensions of the tool set. It is future work to develop more case studies, similar in size and complexity, but taking advantage of the different wrapper tools. For all three verification scenarios appropriate wrappers will be developed. Further, we plan to provide support for other property specification formalisms, in particular security automata. Also, support for flow graph extraction from source code will be improved, developing a modular and extensible tool. Other extensions concern the program model, where we plan to add data to flow graphs to represent Boolean programs faithfully, and to develop a solution for multi–threaded programs. Finally, we plan to extend the logic to include liveness properties; these become meaningful when the flow graphs model program behaviour faithfully, or at least provide under–approximations of the guaranteed behaviour.

Acknowledgements We thank everybody who contributed to CVPP: Irem Aktug (Analyser), Christoph Sprenger (MaxMod), Siavash Soleimanifard (ProMoVer), and Afshin Amighi (property simplification). We are also indebted to Stefan Schwoon, who extended the input language of Moped to serve our needs.

References

1. R. Alur, M. Arenas, P. Barcelo, K. Etessami, N. Immerman, and L. Libkin. First-order and temporal logics for nested words. In *Logic in Computer Science (LICS '07)*, pages 151–160, Washington, DC, USA, 2007. IEEE Computer Society.
2. R. Alur and S. Chaudhuri. Temporal reasoning for procedural programs. In *Verification, Model Checking, and Abstract Interpretation (VMCAI '10)*, volume 5944 of *LNCS*, pages 45–60. Springer, 2010.
3. R. Alur, K. Etessami, and P. Madhusudan. A temporal logic for nested calls and returns. In *Tools and Algorithms for the Analysis and Construction of Software (TACAS '04)*, volume 2998 of *LNCS*, pages 467–481. Springer, 2004.
4. M. Barnett, K.R.M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *CASSIS 2004*, volume 3362 of *LNCS*. Springer, 2004.
5. G. Boudol and K. Larsen. Graphical versus logical specifications. *Theoretical Computer Science*, 106:3–20, 1992.
6. G. Chugunov, L.-Å. Fredlund, and D. Gurov. Model checking of multi-applet JavaCard applications. In *Smart Card Research and Advanced Application Conference (CARDIS '02)*, pages 87–95. USENIX Publications, 2002.
7. R. Cleaveland, J. Parrow, and B. Steffen. A semantics based verification tool for finite state systems. In *International Symposium on Protocol Specification, Testing and Verification*, pages 287–302. North-Holland Publishing Co., 1990.
8. M. Goldman and S. Katz. MAVEN: Modular aspect verification. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '07)*, volume 4424 of *LNCS*, pages 308–322. Springer, 2007.
9. O. Grumberg and D. Long. Model checking and modular verification. *ACM TOPLAS*, 16(3):843–871, 1994.
10. D. Gurov and M. Huisman. Interface abstraction for compositional verification. In *Software Engineering and Formal Methods (SEFM '05)*, pages 414–423, 2005.
11. D. Gurov and M. Huisman. Reducing behavioural to structural properties of programs with procedures. In *Verification, Model Checking, and Abstract Interpretation (VMCAI '09)*, volume 5403 of *LNCS*, pages 136–150. Springer, 2009.
12. D. Gurov and M. Huisman. Reducing behavioural to structural properties of programs with procedures. 2010. Full version, submitted, available upon request.
13. D. Gurov, M. Huisman, and C. Sprenger. Compositional verification of sequential programs with procedures. *Information and Computation*, 206(7):840–868, 2008.
14. M. Huisman, I. Aktug, and D. Gurov. Program models for compositional verification. In *International Conference on Formal Engineering Methods (ICFEM '08)*, volume 5256 of *LNCS*, pages 147–166. Springer, 2008.
15. M. Huisman, D. Gurov, C. Sprenger, and G. Chugunov. Checking absence of illicit applet interactions: a case study. In *Fundamental Approaches to Software Engineering (FASE '04)*, volume 2984 of *LNCS*, pages 84–98. Springer, 2004.
16. D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

17. P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *LNCS*. Springer, 2002.
18. F. B. Schneider. Enforceable security policies. *ACM Trans. Infinite Systems Security*, 3(1):30–50, 2000.
19. S. Schwoon. *Model-Checking Pushdown Systems*. PhD thesis, Technische Universität München, 2002.
20. S. Soleimanifard, D. Gurov, and M. Huisman. Procedure–modular verification of control flow safety properties. In *Workshop on Formal Techniques for Java Programs (FTfJP '10)*, 2010.
21. R. Vallée-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java Optimization Framework. In *CASCON '99*, pages 125–135, 1999.
22. I. Walukiewicz. Completeness of Kozen's axiomatisation of the propositional mu-calculus. In *Logic in Computer Science (LICS '95)*, pages 14–24. IEEE, 1995.

A Pushdown System Representation for Unbounded Object Creation

Jurriaan Rot[1], Frank de Boer[1,2], Marcello Bonsangue[1,2]

[1] LIACS – Leiden University
[2] Centrum voor Wiskunde en Informatica (CWI)
jrot@liacs.nl,frb@cwi.nl,marcello@liacs.nl

Abstract. We introduce a block-structured programming language which supports object creation, global variables, static scope and recursive procedures with local variables. Because of the combination of recursion, local variables and object creation, the number of objects stored during a computation is potentially unbounded. However, we show that a program can be viewed as a type of pushdown automata, for which the halting problem as well as LTL and CTL model checking are decidable.

Key words: object creation, model checking, pushdown systems, pushdown automata

1 Introduction

From the 1960s onwards imperative programming has evolved with the introduction of high-level programming constructs for mastering the complexity of software by abstraction, encapsulation and modularity. The initial description of computation in terms of assignment statements to change a program state, sequential and conditional composition, and conditional looping [10] has been extended with procedures in combination with block structures which enable the construction and declaration of complex state changes abstracting from the concrete implementation [19]. Pointers are a very flexible programming mechanism, allowing manipulation of dynamically growing and potentially unbounded data structures. In the eighties mainstream imperative programming languages added the support of objects [23], a collection of procedures acting on an encapsulated state, having an identity that can be referred to by other objects. Other powerful programming techniques like inheritance and polymorphism enable code reuse, and are crucial for programming-in-the-large [22].

The increasing flexibility in programming comes, however, with an increasing complexity in reasoning about programs. Model checking is a technique for exhaustively checking a (model of a) program for possible errors [4]. Traditionally, in order to guarantee termination of a model checking procedure, finite-state models are required, and thus only programs over finite data domains are considered. But to program with dynamical data structures, objects may need to be created, removed and modified when moving from a state to another in a

computation. Thus, by their very nature, objects are unbounded: for example, during a recursive computation new objects can be created infinitely often. In order to achieve finite-state models for object-oriented programs, different types of abstraction and restrictions of programs have been considered (see related work below). Typically one disallows object creation and considers only a finite number of objects already existing before the computation starts, or allows for object creation only within restricted forms of recursion. However, the necessity to restrict programs before their analysis limits the applicability of model checking techniques to modern imperative programming languages.

In this paper we introduce a simple block-structured programming language which supports object creation, global variables, static scope and recursive procedures with local variables. In order to focus on the main issues, we restrict to a single but unbounded data structure, namely that of object identities. Other *finite* data domains could have been added without problem, but would have increased the complexity of the model without strengthening our main result. Although very simple, the language is powerful enough to encode the control flow of high-level imperative programming languages including closed class-based object-oriented programs, like Java. Because of the combination of recursion with local variables, a program may have infinitely many different states. Since we allow to store object references into local variables, the number of objects stored during a computation is potentially unbounded.

For our language we define two semantics: a concrete one that is infinite state, and a symbolic one that is also infinite state but is based on an enhanced version of the model of recursive procedures with local variables via a suitable pushdown system [12]. A pushdown system is a simple type of pushdown automaton used to generate behavior rather than to accept languages [5]. It provides a finite representation generating infinite state systems, where a state consists of a control part and a stack. In our model of a pushdown system global variables and the current local variables form the control states, whereas the current executing statement is on top of the stack. Actually it is more common to model only the global variables in the control state, while the local variables and the control point are (part of) the top of the stack (see e.g. [21]). We chose our approach for convenience in the proofs, but it can easily be modified to the more common approach. In our model, when a procedure is called, a copy of the current local variables is stored on the stack to recover the original values after the procedure returns, and the local variables in the control state are initialized again. In order to achieve finitely many control states, we abstract from the concrete identities of the objects, but maintain their symmetries, i.e. the equality relation among object identities [13]. Our main result is that the concrete and the symbolic semantics are strongly bisimilar.

Reachability for an infinite state system is generally undecidable. However, for a pushdown system it turns out that both the halting problem and reachability are decidable [16]. In fact, it is possible to model check pushdown systems against linear-time or branching-time temporal formulas. For linear-time

temporal formulas the complexity is even of the same order as for finite state systems [5].

This paper is organized as follows. In Section 2, we introduce the syntax of our language and give an informal description of its semantics. Section 3 provides a concrete execution model using a transition system on infinite states, and in Section 4 we describe the construction for the symbolic semantics based on pushdown systems. The relationship between these two models is studied in Section 5. Finally, the last section discusses some relevant consequences of our result, and possible future steps.

Related work. Currently there are several model checkers for object oriented languages. Java Path Finder [14] is basically a Java Virtual Machine that executes a Java program not just once but in all possible ways, using backtracking and restoring the state during the state-space exploration. Even if Java Path Finder is capable of checking every Java program, the number of states stored during the exploration is a limit on what can be effectively checked. As with JCAT [9], Java source code can be translated into Promela, the input language of SPIN [15]. Since Promela does not support dynamic data structures, they have to allocate fixed-size heaps and stacks.

Bandera [8] is an integrated collection of tools for model-checking concurrent Java software using state-of-the art abstraction, partial order reductions and slicing techniques to reduce the state space. It compiles Java source code into a reduced program model expressed in the input language of other existing verification tools. For example, it can be combined with the SAL (Symbolic Analysis Laboratory) model checker [18] that uses unbounded arrays whose sizes vary dynamically to store objects. In order to explore all reachable states model checking is restricted to Java programs with a bounded (but not fixed a priori) number of objects.

Model checking of a possibly unbounded number of objects but for a language with a restricted form of recursion (tail recursion) and no block structure has been studied using high level allocation Büchi automata [11], a generalization of history dependent automata [17] that enables for a finite state symbolic semantics very similar to ours. Full recursion, but with a fixed-size number of objects is instead considered in jMoped [12], using a pushdown structure to generate an infinite state system.

The current state of the art of model checking approaches for languages with object creation and full recursion in terms of concrete memory addresses, require an a priori bound on the size of the heap for reachability analysis (e.g. [6]). In order to overcome this problem, the main contribution of this paper is the precise abstraction of the heap in terms of equivalence classes of program variables which refer to the same memory address.

2 A simple imperative language with object creation

This section introduces a simple programming language that supports object creation, global and local variables, and recursive procedures. To simplify the

presentation it is restricted to a single data structure, that of object identities. A program consists of a finite set of procedures, each acting on some global and local state. Procedures can store identities in global or local variables, compare them, and call other procedures.

We assume a finite set of *program variables* V ranged over by $x, y, \ldots$ such that $V = G \cup L$, where G is a set of *global variables* $\{g_1, g_2, \ldots, g_n\}$ and L is a set of *local variables* $\{l_1, l_2, \ldots, l_m\}$, with G and L disjoint. For P a finite set of *procedure names* $\{p_0, \ldots, p_k\}$, a program is a set of *procedure declarations* of the form $p_i :: B_i$, where B_i, denoting the *body* of the procedure p_i, is a statement defined by the following grammar

$$B \ ::= \ x := y \mid x := \text{new} \mid B; B \mid [x = y]B \mid [x \neq y]B \mid B + B \mid p\,.$$

Here x and y are program (local or global) variables in V, and p is a procedure name in P. The procedure $p_0 \in P$ is called the *initial* procedure of a program.

The language is statically scoped. The *assignment* statement $x := y$ assigns the identity stored in y (if any) to x. If x was already referring to an object identity, this gets lost. In particular, if x is the only variable of the program referring to an object o, then after an assignment $x := y$, the object o cannot be referenced anymore and gets lost forever. The statement $x :=$ new *creates* a new object that will be referred to by the program variable x. As for the ordinary assignment, the old value of x is lost. In a program execution, a program variable x is said to be *defined* if there was an assignment or object creation statement earlier in the execution with the variable x at left–hand side. *Sequential composition* $B_1; B_2$, *conditional statements* $[x = y]B$ and $[x \neq y]B$ and *nondeterministic choice* $B_1 + B_2$ have the standard interpretation. A *procedure call* p means that the body B associated with p is executed next on the same global state but on a new fresh local state. After the procedure body terminates, its local state is destroyed forever and the previous local state (from which the procedure has been called) is restored. Changes to the global state, however, remain.

More general boolean expressions in conditional statements can be obtained by using sequential composition and nondeterministic choice. In fact $(b_1 \wedge b_2)B$ can be written as $(b_1)b_2B$, whereas $(b_1 \vee b_2)B$ as $(b_1B) + (b_2B)$. Negation of a boolean expression b can be obtained by transforming b into an equivalent boolean expression in conjunctive disjunctive normal form, for which negation of the simple expression $[x = y]$ and $[x \neq y]$ is defined as expected.

Ordinary while, skip, and if-then-else statements can be expressed easily in the language, using recursive procedures, conditional statements and nondeterministic choice. For the sake of simplicity, we allow creation and assignment of a single object identity only; generalizations to simultaneous assignments and object creation can be added in a straightforward manner. We assume automatic garbage collection of object identities that are not referenced anymore by any global variables or instances of local variables in a program execution.

The language does not directly support parameter passing. However, it is worthwhile to note that we can model procedures with call-by-value parameters by means of global variables. Let $p(v_1, \ldots, v_n)$ be a procedure with formal

parameters $v_1, \ldots, v_n$. We see the formal parameters as local variables and introduce for each parameter v_i a corresponding global variable g_i (which does not appear in the given program). Every procedure call $p(x_1, \ldots, x_n)$ can be modeled by the statement $g_1 := x_1; \ldots; g_n := x_n; p$ whereas the body B of $p(v_1, \ldots, v_n)$ can be modeled by $v_1 := g_1; \ldots; v_n := g_n; B$. A similar approach can be taken to model procedures with return values. Finally, method calls $x.m(x_1, \ldots, x_n)$ then can be modeled by introducing the called object x as an additional 'parameter' of the procedure m.

3 Transition System Semantics

In this section, we introduce a semantics of the programming language which is defined in terms of an explicit representation of objects by natural numbers. This representation allows a simple implementation of object creation. A *program state* of a program is a function

$$s : V \longrightarrow \mathbb{N}_\perp,$$

where $\mathbb{N}_\perp = \mathbb{N} \cup \{\perp\}$ ($\perp$ is used to denote "undefined"). To model object creation we distinguish a global "system" variable c which is used as a counter, and is not used by programs. We implicitly assume that $s(c) \neq \perp$, for every state s.

A *configuration* of a program is a pair $\langle s, S\rangle$ where s is a program state and S is a stack of statements and local states. An *execution step* of a program is a transition from a configuration C to a configuration C', denoted by $C \longrightarrow C'$. The possible execution steps are given below. For modeling state updates we use multiple assignments of the form $s[x_1, \ldots, x_n := v_1, \ldots, v_n]$, where x_i and x_j are distinct, for $i \neq j$. The head of a stack is separated from the tail with the right-associative operator $\bullet$; for example, $S' = e \bullet S$ is the stack consisting of head e and tail S.

$$\langle s, B_1; B_2 \bullet S\rangle \longrightarrow \langle s, B_1 \bullet B_2 \bullet S\rangle \tag{1}$$

$$\frac{s(y) \neq \perp}{\langle s, x := y \bullet S\rangle \longrightarrow \langle s[x := s(y)], S\rangle} \tag{2}$$

$$\langle s, x := \text{new} \bullet S\rangle \longrightarrow \langle s[x, c := c, c+1], S\rangle \tag{3}$$

$$\frac{s(x) = s(y) \qquad s(x) \neq \perp}{\langle s, [x = y]B \bullet S\rangle \longrightarrow \langle s, B \bullet S\rangle} \tag{4}$$

$$\frac{s(x) \neq s(y) \qquad s(x) \neq \perp \qquad s(y) \neq \perp}{\langle s, [x \neq y]B \bullet S\rangle \longrightarrow \langle s, B \bullet S\rangle} \tag{5}$$

$$\langle s, B_1 + B_2 \bullet S\rangle \longrightarrow \langle s, B_i \bullet S\rangle \quad (i \in \{1, 2\}) \tag{6}$$

$$\langle s, p_i \bullet S\rangle \longrightarrow \langle s', B_i \bullet s \bullet S\rangle \tag{7}$$

where $s'(l) = \bot$, for every local variable l and $s'(g) = s(g)$, for every global variable g.

$$\langle s, s' \bullet S\rangle \longrightarrow \langle s[\bar{l} := s'(\bar{l})], S\rangle \tag{8}$$

where $\bar{l}$ denotes the sequence of local variables $l_1, \ldots, l_m$ and $s'(\bar{l})$ denotes the sequence of values $s'(l_1), \ldots, s'(l_m)$.

For technical convenience only, a procedure call pushes onto the stack as local environment the entire state. Further, we assume a distinguished global variable 'nil' such that $s(\text{nil}) = \bot$, for every state s. The following corollary states some basic properties of the semantic rule 8.

Corollary 1. *If $\langle s, s' \bullet S\rangle \longrightarrow \langle s'', S\rangle$ then for every $x, y \in V$:*

1. *x and y are both global implies $s''(x) = s''(y)$ iff $s(x) = s(y)$,*
2. *x and y are both local implies $s''(x) = s''(y)$ iff $s'(x) = s'(y)$,*
3. *x is global and y is local implies $s''(x) = s''(y)$ iff $s(x) = s'(y)$.*

Proof. It suffices to observe that by definition of the rule 8 we have $s''(g) = s(g)$ and $s''(l) = s'(l)$, for every global g and every local l. □

Further, we have the following invariance property about the flow of information between the current state and the stacked states.

Lemma 1. *For every computation $\langle s_0, p_0\rangle \longrightarrow^* \langle s, S\rangle$, variable z, local variable l and local state s' appearing in S, we have $s(z) = s'(l)$ iff there exists a global variable g such that $s(z) = s'(g)$ and $s'(l) = s'(g)$.*

Proof. The proof is by induction on the length of the computation. The basis of the induction is trivial because the stack of the initial configuration only contains the statement p_0.

It suffices to show that every production respects the property. First note that for any rule except 7, no new states are added to the stack so we only need show that the resulting state still satisfies the equivalence. In rule 1, 4, 5 and 6 we have $s = s'$ so the equivalence holds by the induction hypothesis. In the assignment rule 2, the resulting state is $s[x := s(y)]$. Now if $z \equiv x$ ($\equiv$ denotes *syntactic* identity) then $s[x := s(y)](z) = s(y)$ and if $z \not\equiv x$ then $s[x := s(y)](z) = s(z)$. In both cases, the result follows from the induction hypothesis. In rule 3, the resulting state is $s[x, c := c, c + 1]$. It follows that $s[x, c := c, c + 1](y) = s(y)$ and $s[x, c := c, c + 1](x) \neq s(y)$, for all $y \not\equiv x$. The result then follows from the induction hypothesis. Rule 7, the procedure call, adds the state s to the top of the stack. Since the values of the locals are $\bot$ in the resulting state, and the values of globals in the resulting state are equal to their value in s, by the induction hypothesis the equivalence holds for every state in the resulting stack including s. Lastly for rule 8 does not alter the globals. For the locals, the result follows from the induction hypothesis for the popped abstract state. Note that there exists a computation $\langle s_0, p_0\rangle \longrightarrow \langle s', S\rangle$, where s' denotes the popped abstract state. □

4 Pushdown System Semantics

The above semantics gives rise to an infinite state system because of unbounded recursion, and because of the representation of objects by natural numbers used to model unbounded object creation. Since we can only test objects for equality we can reduce this state-space by the introduction of equivalence classes of variables, that is, two variables belong to the same equivalence class if they denote the same object. However local variables can generate again an unbounded number of equivalence classes. We show in this section how we can restrict to an *apriori finite* number of equivalence classes of variables by the introduction of so-called "freeze" variables, which will be used to compare the partitions of variables before and after executing a procedure call. This will allows for a reallocation of the global variables with respect to the local variables of the caller. To do this, we associate with each global variable g a fresh and unique local variable g' (which we assume does not appear in the given program).

An *abstract* program state now consists of a partition of global and local variables (including the freeze variables). To facilitate easy treatment of such a partition, we represent it as a function

$$\sigma : V \longrightarrow |V| + 1$$

where $|V|$ is the cardinality of the set of variables V, and $|V| + 1$ is identified with the set $\{0, \ldots, |V|\}$. Thus two (distinct) variables x and y belong to the same equivalence class iff $\sigma(x) = \sigma(y)$. We use zero for the equivalence class of variables which are undefined, e.g., $\sigma(\text{nil}) = 0$, for every abstract state σ.

A *configuration* of a program now is a pair $\langle \sigma, \Sigma \rangle$ where σ is an abstract state as defined above and Σ is a stack of statements and abstract states. Because of the way we model partitions of the set of variables V, rules 1, 2, 4, 5 and 6 directly apply in this model and are therefore not repeated here. The rule for object creation is modified as follows.

$$\langle \sigma, x := \text{new} \bullet \Sigma \rangle \longrightarrow \langle \sigma', \Sigma \rangle \tag{9}$$

where $\sigma' = \sigma$, if all indices except zero are used in σ, else $\sigma' = \sigma[x := i]$, where $i \neq 0$ is the smallest index not already used by σ.

This new rule for object creation describes it in terms of an update of the current partition of the variables V by isolating the variable x. This is achieved by assigning to the variable x an index different from zero not in use. Note that in case such an index does not exist the partition represented by σ consists of singleton sets only and therefore is not affected by object creation, i.e., we do *not* need to assign a new index to x because it is already isolated.

The rule for procedure calls is modified as follows.

$$\langle \sigma, p_i \bullet \Sigma \rangle \longrightarrow \langle \sigma', B_i \bullet \sigma \bullet \Sigma \rangle \tag{10}$$

where $\sigma' = \sigma[\overline{l} := \overline{0}][\overline{g'} := \sigma(\overline{g})]$, $\overline{g'}$ denotes the sequence $g'_1, \ldots, g'_n$ of freeze variables and $\sigma(\overline{g})$ denotes the sequence of indices $\sigma(g_1), \ldots, \sigma(g_n)$. Note that $\sigma[\overline{l} := \overline{0}](l) = 0$, for every local variable $l \in L$.

A procedure call now additionally initializes the freeze variables by the values of their corresponding global variables and stores the old abstract state onto the stack. Note that execution of B_i does not affect the freeze variables.

Finally, the rule for returns from a procedure call is modified as follows.

$$\langle \sigma, \sigma' \bullet \Sigma \rangle \longrightarrow \langle \sigma_n, \Sigma \rangle \tag{11}$$

where $\sigma_0 = \sigma'$ and for $0 < i \leq n$ (where n is the number of globals) we define σ_i by the following cascade of if-then-else statements:

- if $\sigma(g_i) = 0$ then $\sigma_i = \sigma_{i-1}[g_i := 0]$ else
- if $\sigma(g_i) = \sigma(g_j)$, for some $j < i$, then $\sigma_i = \sigma_{i-1}[g_i := \sigma_{i-1}(g_j)]$ else
- if $\sigma(g_i) = \sigma(g')$, for some freeze variable g', then $\sigma_i = \sigma_{i-1}[g_i := \sigma'(g)]$ else
- if in σ_{i-1} all indices except 0 are used then $\sigma_i = \sigma_{i-1}$ else
- $\sigma_i = \sigma_{i-1}[g_i := k']$, where $k' \neq 0$ is the smallest index not already used by σ_{i-1}.

Upon return, which consitutes the 'heart of the matter', we need to update the stored partition σ' by reallocating the global variables according to the new partition described by σ. We do so by means of the freeze variables which represent in σ the partitioning of the global variables in σ' and as such form a reference point for comparison with the local variables in σ'. In other words, a partition in σ' containing global variables is represented in σ by the corresponding freeze variables. Therefore, in case in σ a global variable g_i is identified with a freeze variable g' we have to identify it with all the local variables which belong to the partition of g in σ'. This is simply obtained by setting the index of g_i to $\sigma'(g)$. Note that in fact $\sigma'(g) = \sigma(g')$. However, $\sigma'(g) = \sigma'(g')$ does *not* hold in general because the freeze variable g' represents the *initial* value of its global variable g which may have been affected by the computation which led to σ'. Further, we observe that the choice of a particular freeze variable does not affect the reallocation because if two distinct freeze variables are identified in σ, then so are their corresponding global variables in σ'. Finally, we note that global variables which are "drifted away" from these freeze variables can only denote objects which are different from those denoted by the local variables in σ'. Therefore for these variables new partitions have to be created. In order to obtain suitable indices for these global variables we have defined the overall update of σ' incrementally by processing the global variables one by one. For each global variable g_i its reallocation is defined by σ_i as follows: if g_i is undefined in σ then so it is in σ_i, else if g_i is identified by σ with some already processed g_j $(j < i)$ then we set its index to that of g_j in σ_{i-1}, else if g_i is identified by σ with some freeze variable then we set its index to that of the corresponding global variable in σ'. In case none of the above holds then we have to create a new partition for g_i as in the new rule for object creation.

Example 1. We give an example of a derivation which illustrates the procedure call and return. The state is represented as a partition. We assume p is a procedure name with body $p :: B = g_2 := \text{new}$. Furthermore g_1, g_2 are global variables, l_1, l_2 are local variabes.

$$
\begin{array}{ll}
 & \langle\{\{g_1, l_1\}, \{g_2, l_2\}\}, p \bullet \Sigma\rangle \longrightarrow \\
\text{(call)} & \langle\{\{g_1, g_1'\}, \{g_2, g_2'\}, \{l_1, l_2\}\}, g_2 := \text{new} \bullet \{\{g_1, l_1\}, \{g_2, l_2\}\} \bullet \Sigma\rangle \longrightarrow \\
\text{(creation)} & \langle\{\{g_1, g_1'\}, \{g_2'\}, \{g_2\}, \{l_1, l_2\}\}, \{\{g_1, l_1\}, \{g_2, l_2\}\} \bullet \Sigma\rangle \longrightarrow \\
\text{(return)} & \langle\{\{g_1, l_1\}, \{g_2\}, \{l_2\}\}, \Sigma\rangle
\end{array}
$$

The first transition step pushes the current state unto the stack. The new state separates all the local variables l_1 and l_2 (this set is indexed by zero which indicates "undefinedness") and introduces the freeze variables. The execution of $g_2 := \text{new}$ in the next transition step isolates the variable g_2. Finally, upon returning, g_1 is still identified with l_1 but both l_2 and g_2 are now isolated. It is important to note that in the above computation we can also replace l_1 and l_2 by freeze variables of earlier procedure calls.

Each set of variables identified by an abstract state defines an object. Further, as explained above, two sets of variables V_i and V_{i+1} identified by the respective abstract states σ_i and σ_{i+1}, which are stored consecutively (from bottom to top) on a given stack Σ, define the same object if and only if there exists a global variable $g \in V_i$ for which its freeze variable g' is in V_{i+1}. The equivalence relation induced by this relation between the sets of variables stored on a given a stack Σ represents the objects generated by Σ. Figure 1 depicts a chain of sets of variables which denote the same object.

The following corollary states some basic properties of the semantic rule 8.

Corollary 2. *If* $\langle\sigma, \sigma' \bullet \Sigma\rangle \longrightarrow \langle\sigma'', \Sigma\rangle$ *then for every* $x, y \in V$*:*

1. *x and y are both global implies $\sigma''(x) = \sigma''(y)$ iff $\sigma(x) = \sigma(y)$*
2. *x and y are both local implies $\sigma''(x) = \sigma''(y)$ iff $\sigma'(x) = \sigma'(y)$*
3. *x is global and y is local implies $\sigma''(x) = \sigma''(y)$ iff there exists a global variable g such that $\sigma'(y) = \sigma'(g)$ and $\sigma(x) = \sigma(g')$*

Proof. The equivalences follow immediately from the construction of σ'', stated in rule 8. □

Clearly the above semantics can be represented as a *pushdown system* (PDS). A pushdown system is a triplet $\mathcal{P} = (Q, \Gamma, \Delta)$ where Q is a finite set of *control locations*, Γ is a finite *stack alphabet*, and $\Delta \subseteq (Q \times \Gamma) \times (Q \times \Gamma^*)$ is a finite set of *productions*. A transition $(q, \gamma, q', \bar{\gamma})$ is enabled if control is at location q and γ is at the top of the stack then control can move to location q' by replacing γ by the possible empty work of stack symbols $\bar{\gamma}$.

In our case, for a given program $p_1 :: B_1, \ldots, p_n :: B_n$, the set of control locations is defined by the finite abstract state space $V \longrightarrow |V| + 1$. In order to define the stack alphabet we introduce the finite set $\bigcup_{i=1}^{k} cl(B_i)$ of possible reachable statements where the closure of a statement B, denoted as $cl(B)$, is defined as follows.

- $cl(x := y) = \{x := y\}$
- $cl(x := \text{new}) = \{x := \text{new}\}$
- $cl(B; B) = cl(B) \cup cl(B)$

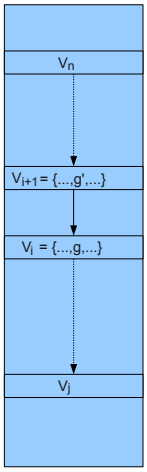


Fig. 1. Chain in a Stack

- $cl([x = y]B) = \{[x = y]B\} \cup cl(B)$
- $cl([x \neq y]B) = \{[x \neq y]B\} \cup cl(B)$
- $cl(B + B) = cl(B) \cup cl(B)$
- $cl(p) = \{p\}$

The stack alphabet Γ is then defined by the union of the abstract state space and the above set of possible reachable statements. Finally, it is straightforward to transform the rules of the above semantics into rules of a pushdown system, simply by removing the common stack tail from the left- and righthand sides.

5 Equivalence between the two models

In this section the behavioural equivalence between the two models is shown by establishing *bisimilarity*, which is widely accepted as the finest behavioural equivalence one would want to impose. A (binary) symmetric relation $\mathcal{R}$ on the states of a transition system which satisfies

$$\text{if } P \longrightarrow P' \text{ then there is a } Q' \text{ such that } Q \longrightarrow Q' \text{ and } (P', Q') \in \mathcal{R},$$

is called a *bisimulation relation* [20].

This definition applies to a single transition system – in our case, we use it to establish equivalence between the two models. The states of the transition system are pairs of configurations, and the transitions are execution steps of the respective models.

We first define the following relation between abstract and concrete states.

Definition 1. *We define* $s \sim \sigma$ *by* $s(x) = s(y)$ *iff* $\sigma(x) = \sigma(y)$*, for every pair of variables* x *and* y*.*

Next we extend this relation to stacks and configurations as follows.

Definition 2. *We define* $S \sim \Sigma$ *inductively by*

- *if* S *and* Σ *are both empty then* $S \sim \Sigma$
- *if* $S \sim \Sigma$ *then* $B \bullet S \sim B \bullet \Sigma$*, for any statement* B
- *if* $s \sim \sigma$ *and* $S \sim \Sigma$ *then* $s \bullet S \sim \sigma \bullet \Sigma$

We define $\langle s, S\rangle \sim \langle \sigma, \Sigma\rangle$ *by* $s \sim \sigma$ *and* $S \sim \Sigma$*.*

In order to prove equivalence of the concrete and abstract semantics, we introduce the freeze variables also as *auxiliary* variables into the concrete semantics. We do so by implicitly assuming that the rule for procedure calls additionally initializes each freeze variable to the value of its corresponding global variable. Note that this does not affect the behaviour of the program (which is assumed not to contain freeze variables). It therefore suffices to relate this concrete semantics extended with freeze variables and the abstract semantics.

Theorem 1. *The above relation* $\langle s, S\rangle \sim \langle \sigma, \Sigma\rangle$ *is a bisimulation relation for* reachable *configurations* $\langle s, S\rangle$ *for which there exists an initial configuration* $\langle s_0, p_0\rangle$ *such that* $\langle s_0, p_0\rangle \longrightarrow^* \langle s, S\rangle$*.*

Proof. Let $\langle s, S\rangle \sim \langle \sigma, \Sigma\rangle$, where $\langle s, S\rangle$ is a reachable configuration. We must show that for every execution step applicable to one configuration, there is an execution step for the other configuration such that the resulting configurations are again related by $\sim$.

If the top of the stack is any statement except $S + S$, it uniquely determines the next step for both models. We choose the same step for both models for the case $S + S$, so we can consider the resulting configurations of applying an execution step to both configurations. If the execution steps have preconditions (rules 2, 4, 5) then satisfaction of these preconditions must be equivalent in s and σ. It is easy to see this follows from the definition of the relation $\sim$ on states. Now we can establish execution steps $\langle s, S\rangle \longrightarrow \langle s', S'\rangle$ and $\langle \sigma, \Sigma\rangle \longrightarrow \langle \sigma', \Sigma'\rangle$. It rests to prove that the resulting configurations are again equivalent –

$$\langle s', S'\rangle \sim \langle \sigma', \Sigma'\rangle$$

must hold.

We prove the equivalence by considering all semantic rules. We consider the main rules for object creation, procedure calls and returns.

Rule 3 ($z := \text{new}$). For variables x and y distinct from z, we have $s'(x) = s(x)$, $\sigma'(y) = \sigma(y)$, $s'(x) \neq s'(z)$ and $\sigma'(x) \neq \sigma'(z)$. This proves $s' \sim \sigma'$. Next observe that S' and Σ' equals S and Σ, respectively. So we obtain the desired result.

Rule 7 (call p). By definition we have $s'(l) = \bot$ and $\sigma'(l) = 0$, for every local variable l, and $s(g) = s'(g)$ and $\sigma(g) = \sigma'(g)$, for every global variable g. It

follows that $s \sim \sigma$ implies $s' \sim \sigma'$. Further, by definition S' and Σ' equals $s \bullet S$ and $\sigma \bullet \Sigma$, respectively. By assumption, $s \sim \sigma$ and $S \sim \Sigma$, and so by definition $s \bullet S \sim \sigma \bullet \Sigma$.

Rule 8. By definition S and Σ equals $s'' \bullet S'$ and $\sigma'' \bullet \Sigma'$, respectively, for some states s'' and σ''. From the assumption $S \sim \Sigma$ it thus follows that $s'' \sim \sigma''$ and $S' \sim \Sigma'$. Remains to prove that $s' \sim \sigma'$. We distinguish the following three cases:

1. x and y are both global variables:

$$\begin{aligned} s'(x) = s'(y) &\overset{\text{(Corollary 1.1)}}{\iff} s(x) = s(y) \\ &\overset{\text{(Assumption)}}{\iff} \sigma(x) = \sigma(y) \\ &\overset{\text{(Corollary 2.1)}}{\iff} \sigma'(x) = \sigma'(y) \end{aligned}$$

2. x and y are both local variables:

$$\begin{aligned} s'(x) = s'(y) &\overset{\text{(Corollary 1.2)}}{\iff} s''(x) = s''(y) \\ &\overset{\text{(Assumption)}}{\iff} \sigma''(x) = \sigma''(y) \\ &\overset{\text{(Corollary 2.2)}}{\iff} \sigma'(x) = \sigma'(y) \end{aligned}$$

3. x is global and y is local:

$$\begin{aligned} s'(x) = s'(y) &\overset{\text{(Corollary 1.3)}}{\iff} s(x) = s''(y) \\ &\overset{\text{(Lemma 1)}}{\iff} s(x) = s''(g) \text{ and } s''(g) = s''(y), \text{ for some global variable } g \\ &\overset{\text{(Freeze var.)}}{\iff} s(x) = s(g') \text{ and } s''(g) = s''(y), \text{ for some global variable } g \\ &\overset{\text{(Assumption)}}{\iff} \sigma(x) = \sigma(g') \text{ and } \sigma''(g) = \sigma''(y) \text{ for some global variable } g \\ &\overset{\text{(Corollary 2.3)}}{\iff} \sigma'(x) = \sigma'(y) \end{aligned}$$

   Note that because of the introduction of freeze variables in the concrete semantics we indeed have $s''(g) = s(g')$ (this can be proved in a straightforward manner by induction on the length of the computation).

This concludes the proof of Theorem 1. □

6 Conclusions

Pushdown systems naturally model the control flow of sequential computation in programming languages with local variables and recursive procedures. In this paper we provided a generalization of this model by adding unbounded object creation. We have shown that imperative programs with object creation, recursive procedures, and local variables without any restriction can be given a symbolic semantics through a finite pushdown system such that the infinite state system generated is strongly bisimilar to the ordinary operational semantics of the program.

Applications to static analysis. Starting from an initial stack containing the initial procedure p_0, a program P is executed and eventually terminates when the stack is empty. If we consider a singleton alphabet symbol labeling all transitions of our pushdown system, we obtain an ordinary pushdown automaton (with acceptance by empty stack). Clearly, the language accepted by this pushdown automaton is non-empty if and only if there exists an execution of the program P that terminates. Since the emptiness problem is decidable for pushdown automata [16], we have an algorithm for deciding termination of programs in our language. Similarly, because the halting problem for pushdown automata is decidable, we have an algorithm for deciding if a program blocks, for example because of an assignment with an undefined variable at the right-hand side.

Applications to model checking. More recently, the problem of checking ω-regular properties (like those expressible in linear-time temporal logics or linear-time μ-calculus) or properties expressed as formulas of the alternation-free modal μ-calculus (including CTL properties) of pushdown systems have been shown to be decidable, leading to efficient model checkers for the generated infinite state systems (see e.g. [5,12]). For instance, to verify whether a program P in our language satisfies a linear time temporal formula ϕ, we first derive a symbolic pushdown system for P with finitely many control states and stack symbols, then construct the finite state Büchi automaton for the negation of ϕ, and finally use the algorithm of [5] to check if there is no execution of the program P that satisfies the negation of ϕ. Interestingly, the complexity of this model checking problem for a fixed LTL formula is polynomial in the size of the pushdown system, a complexity that is not much worse than that for finite transition systems [5].

In the future we plan to investigate the integration of our technique with jMoped, a Java model checker based on pushdown systems [12]. As for the model checking, there are at least two directions that could be explored. On the one hand we intend to look for extension of temporal logic with support for a primitive for object creation (and destruction) [11,3]. On the other hand, we would like to investigate model checking of some non ω-regular properties, allowing, for example, matching of procedure calls and returns. While the problem is in general undecidable, it seems possible to turn our pushdown systems into visibly pushdown automata, a class of pushdown automata with desirable closure properties and interesting tractable decision problems [1].

Language considerations. We have presented a language that supports unbounded object creation by using recursive procedures with global and local variables. The language can not be extended with higher-order features like passing procedures and internal procedures as parameters of procedure calls, as well as it cannot include features like call-by-name parameter passing because the halting problem for these two class of programs is known to be undecidable [7]. It would be interesting, however, to see what happens if we change static scope to dynamic scope or if we disallow internal procedures as parameters.

Our language does not have any concrete data but for object identities, and does not support object fields. Data can be added but in order to model computations by a *finite* pushdown system, we need to consider only finite data domains. The language can be extended with object fields $f_1, \ldots, f_n$, by simply adding expressions of the form $x.f$ as variables, and as such they will be included in the partitions. More general navigation expressions can be reduced to the above in the obvious way.

The language does not have a syntactic construct to destroy object identities. We can give a concrete semantic for it without the needs of inspecting the call-stack (for example by storing in extra variables the names of the objects destroyed and assuming they will not be reused, so that local variables in the stack can be reset when a procedure returns). This observation can be combined with the concept of chains in the stack of variables referring to the same object to allow deletion within the pushdown system representation, by simply keeping track of the chains which refer to deleted objects. This way of deletion would work also for encoded object fields, which implies on-the-fly garbage collection. We plan to work out the details in a future work.

Finally, our language is sequential. It is not a problem to add bounded concurrency within the body of a procedure by using, e.g. a parallel operator of the form $B_1||B_2$, as we can give an interleaving semantic to it using rules of the form

$$\frac{\langle s, B_1 \bullet S\rangle \longrightarrow \langle s, B \bullet S\rangle}{\langle s, B_1||B_2 \bullet S\rangle \longrightarrow \langle s, B||B_2 \bullet S\rangle} \quad \text{and} \quad \frac{\langle s, B_2 \bullet S\rangle \longrightarrow \langle s, B \bullet S\rangle}{\langle s, B_1||B_2 \bullet S\rangle \longrightarrow \langle s, B_1||B \bullet S\rangle}$$

However for more global notions of concurrency, like threads, we need to store the local variables of the program for each thread. Therefore, to keep the stack alphabet and the number of control states finite in our pushdown system, we have to restrict to a bounded number of threads [2]. It can be interesting to combine our results with those of [6], so to allow reachability analysis of multithreaded programs.

We leave these considerations for future work.

References

1. R. Alur, P. Madhusudan. Visibly pushdown languages. In *Proc. of Annual ACM Symposium on Theory of Computing (STOC 2004)*, pages 202-211, ACM, 2004.
2. F.S. de Boer and I. Grabe. Automated Deadlock Detection in Synchronized Reentrant Multithreaded Call-Graphs. In *Proc. of 36th Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2010)*, volume 5901 of *Lecture Notes in Computer Science*, pages 200-211, Springer, 2010.
3. M.M. Bonsangue, A.Kurz. Pi-Calculus in Logical Form, In *Proc 22nd Annual IEEE Symposium on Logic in Computer Science (LICS 2007)*, pp. 303-312, IEEE, 2007.
4. C. Baier and J.-P. Katoen. *Principles of Model Checking* The MIT press, 2008.
5. A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking In *Proceedings Concur 97*, volume 1243 of *Lecture Notes in Computer Science*, pp. 135–150, Springer, 1997.

6. A. Bouajjani, S. Fratani, S. Qadeer. Context-Bounded Analysis of Multithreaded Programs with Dynamic Linked Structures. In *Proc. Intern. Conf. on Computer Aided Verification (CAV'07* volume 4590 of *Lecture Notes in Computer Science*, Springer 2007.
7. E.M. Clarke. Programming language constructs for which it is impossible to obtain good Hoare-like axioms. *Journal of the ACM* 26:126-147, 1979.
8. J. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *Proceedings 22nd International Conference on Software Engineering*, pp. 439-448. IEEE Computer Society, 2000.
9. C. Demartini, R. Iosif, and R. Sisto. A deadlock detection tool for concurrent Java programs. *Software - Practice and Experience*, 29(7):577–603, 1999.
10. E.W. Dijkstra. *A Discipline of Programming.* Prentice-Hall Series in Automatic Computation, 1976.
11. D. Distefano, J.-P. Katoen, A. Rensink. Who is Pointing When to Whom? In *Proceedings of 24th Int. Conf. on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2004)* volume 3328 of *Lecture Notes in Computer Science*, pp. 250-262, Springer 2004.
12. J. Esparza and S. Schwoon. A BDD-based model checker for recursive programs. In *Proceedings of CAV 2001*, volume 2102 of *Lecture Notes in Computer Science*, pp. 324–336, Springer, 2001.
13. M. Gabbay and A. Pitts. A new approach to abstract syntax involving binders. In *Proceedings of 14th LICS*, pp. 214-224, IEEE Computer Society Press, 1999.
14. K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366-381, 2000.
15. G.J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–94, 1997.
16. J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 3rd edition, 2006.
17. U. Montanari and M. Pistore. An Introduction to History Dependent Automata. In *Proceeding 2nd Workshop on Higher-Order Operational Techniques in Semantics*, volume 10 of *Electonic Notes in Theoretical Computer Science*, pp. 170-188, Elsevier, 1998.
18. D. Park, U. Stern, J. Skakkebaek, and D. Dill. Java Model Checking In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering.* pp. 253–256. IEEE, 2000.
19. B. Randell and L.J. Russell. *ALGOL 60 Implementation: The Translation and Use of ALGOL 60 Programs on a Computer.* Academic Press, 1964.
20. D. Sangiorgi. On the bisimulation proof method. *Mathematical Structures in Computer Science* 8(5):447-479, 1998.
21. S. Schwoon. Model-checking pushdown systems. PhD thesis, Technische Universität München, 2002.
22. R. Sebesta. *Concepts of Programming Languages.* Addison-Wesley, 9th edition, 2009.
23. B. Stroustrup *The C++ Programming Language.*

Validating Timed Models of Deployment Components with Parametric Concurrency *

Einar Broch Johnsen, Olaf Owe, Rudolf Schlatte, and S. Lizeth Tapia Tarifa

Department of Informatics, University of Oslo, Norway
{einarj,olaf,rudi,sltarifa}@ifi.uio.no

Abstract. The concurrent object model has advantages to thread-based concurrency with respect to compositionality and verification. In the concurrent object model, each object conceptually encapsulates a processor. When concurrent objects are deployed, the available processing resources are naturally more restricted. This paper proposes an abstract model of deployment components in terms of concurrent object groups with a restricted number of concurrent processing resources. Deployment components are parametric in the amount of concurrency they provide; i.e., they vary in the number of processor resources. We give a formal semantics of deployment components in rewriting logic, extending the semantics of Creol, and characterize equivalence between deployment components which differ in concurrent resources in terms of test suites. Our semantics is executable on Maude, which allows simulations and test suites to be applied to a deployment component with different concurrency resources.

1 Introduction

Software systems today are increasingly developed to be highly configurable. A development method which attempts to systematize this variability, is software product line engineering [23]; in a product line, different software systems (or products) may be instantiated with different features. To illustrate this approach to software development, consider software for cell phones. Products for different cell phones and service subscriptions are produced by selecting among features such as call forwarding, answering machine, text messaging, etc. In addition to this software variability, products often need to be adapted to different hardware or deployment scenarios. Examples of such variability are found in operating systems, which can be adapted to specific hardware and even to the different numbers of available kernels; web shops, which are deployed on a varying number of servers and may even dynamically perform load balancing between these servers; and information systems within, e.g., healthcare or finance, which may run on a single computer, in a distributed set-up, or even on the cloud. Software product lines raise new challenges for the performance analysis of component-based applications [27]. In this paper, we apply performance analysis to models

* Partly funded by the EU project FP7-231620 HATS: Highly Adaptable and Trustworthy Software using Formal Methods (http://www.hats-project.eu).

of object-oriented components or systems in deployment scenarios which vary in the amount of concurrent resources they can provide to the given component.

This work is based on Creol [10,18], a modeling language for distributed concurrent objects which communicate by asynchronous method calls and futures. Creol's operational semantics is given in rewriting logic [20] and is executable on Maude [9]. Concurrent objects are reminiscent of Actors [2] and Erlang [4]: Objects are inherently concurrent, conceptually each object has a dedicated processor, and there is at most one activity in an object at any time. This concurrency model has attracted attention as an alternative to multi-thread concurrency in object-orientation (e.g., [6]), and been integrated with, e.g., Java [26] and Scala [13]. Concurrent objects support compositional verification of concurrent software [3,10], in contrast to multi-threading [1]. A particular feature of Creol is its cooperative scheduling of method activations inside concurrent objects. Recently, Creol's notion of cooperative scheduling and asynchronous method calls has been integrated in Java by means of concurrent object groups [24].

This paper generalizes the idea of concurrent object groups to *deployment components* which are parametric in the amount of concurrent activity they allow within a time interval. Creol is extended with notions of timed execution and deployment components, which are integrated into Creol's operational semantics. This integration is non-trivial in that it must capture parametric concurrent activities within time frames in terms of an interleaving semantics in order to execute the models in Maude. We characterize the equivalence of different deployment scenarios, varying in the concurrency resources of the deployment components, in terms of test suites of timed observable behavior and use Maude to run tests for our models. This allows the timed behavior of concurrent object models under restricted concurrency assumptions to be validated and compared.

Paper overview. Sect. 2 presents a timed version of Creol, and Sect. 3 the deployment components with parametric concurrency. Sect. 4 illustrates the language by an example. Sect. 5 explains the operational semantics of timed Creol in terms of rewriting logic. Sect. 6 presents testing and simulation results in the context of the example, Sect. 7 discusses related work, and Sect. 8 concludes.

2 Concurrent Objects in Creol

Creol is an abstract behavioral modeling language for distributed active objects, based on asynchronous method calls and processor release points. In Creol, objects conceptually have dedicated processors and live in a distributed environment with asynchronous and unordered communication between objects. Communication is between named objects by means of asynchronous method calls; these may be seen as triggers of concurrent activity, resulting in new activities (processes) in the called object. This section briefly introduces Creol (for further details see, e.g., [10, 18]). Objects are dynamically created instances of classes, declared attributes are initialized to some arbitrary type-correct values. An optional *init* method may be used to redefine the attributes. Active behavior, triggered by an optional *run* method, is interleaved with passive behavior,

Syntactic categories.	*Definitions.*
C, I, m in Names	$IF ::= \texttt{interface}\ I\ \{\ [\overline{Sg}]\ \}$
g in Guard	$CL ::= \texttt{class}\ C\ [(\overline{I\ x})]\ [\texttt{implements}\ \overline{I}]\ \{\ [\overline{I\ x};]\ \overline{M}\}$
s in Stmt	$Sg ::= I\ m\ ([\overline{I\ x}])$
x in Var	$M ::= Sg == [\overline{I\ x};]\ \{\ s\ \}$
e in Expr	$g ::= b \mid x? \mid g \wedge g \mid g \vee g$
b in BoolExpr	$s ::= s; s \mid x := e \mid \texttt{release} \mid \texttt{await}\ g \mid x.\texttt{get} \mid \texttt{return}\ e$
	$\mid \texttt{if}\ b\ \texttt{then}\ \{\ s\ \}\ [\texttt{else}\ \{\ s\ \}] \mid \texttt{while}\ b\ \{\ s\ \} \mid \texttt{skip}$
	$e ::= x \mid b \mid \texttt{new}\ C(\overline{e}) \mid [e]!m(\overline{e}) \mid x.\texttt{get} \mid \texttt{this} \mid \texttt{null} \mid \texttt{now}$

Fig. 1. The syntax of core Timed Creol. Terms such as $\overline{e}$ and $\overline{x}$ denote lists over the corresponding syntactic categories and square brackets denote optional elements.

triggered by method calls. Thus, an object has a set of processes to be executed, which stem from method activations. Among these, at most one process is *active* and the others are *suspended* on a process queue. The scheduling of processes is by default non-deterministic, but controlled by *processor release points* in a cooperative way. Creol is strongly typed: for well-typed programs, invoked methods are supported by the called object (when not *null*), such that formal and actual parameters match. This paper assumes that programs are well-typed.

Figure 1 gives the syntax for a core subset of Timed Creol (omitting, e.g., inheritance). A *program* consists of interface and class definitions and a `main` method to configure the initial state. IF defines an interface with name I and method signatures Sg. A class implements a list $\overline{I}$ of interfaces, specifying types for its instances. CL defines a class with name C, interfaces $\overline{I}$, class parameters and state variables x (of type I), and methods M. (The *attributes* of the class are both its parameters and state variables.) A method signature Sg declares the return type I of a method with name m and formal parameters $\overline{x}$ of types $\overline{I}$. M defines a method with signature Sg and a list of local variable declarations $\overline{x}$ of types $\overline{I}$ and a statement s. Statements may access class attributes, locally defined variables, and the method's formal parameters.

Statements. Assignment $x := e$, sequential composition $s_1; s_2$, **skip**, **if**, **while**, and **return** e constructs are standard. The statement **release** unconditionally release the processor by suspending the active process. In contrast, the guard g controls processor release in the statement **await** g, and consists of Boolean conditions that may contain attributes and return tests $x?$ (see below). If g evaluates to false, the current process is *suspended* and the execution thread becomes idle. In this case, any enabled process may be chosen from the pool of suspended processes. The scheduling of processes is *cooperative* in the sense that processes explicitly yield control and execution in one process may enable the further execution in another. Explicit signaling is redundant.

Expressions e include declared variables x, Boolean expressions b, and object creation **new** $C(\overline{e})$. The specially reserved read-only variable **this** refers to the identifier of the object and **now** refers to the current clock value (explained below). Note that remote access to attributes is not allowed. (The full language includes a functional expression language with standard operators for data types

such as strings, integers, lists, sets, maps, and tuples. These are omitted in the core syntax, and explained when used in the examples.)

Communication in Creol is based on asynchronous method calls, denoted $e!m(\overline{e})$, and future variables. (Local calls are written $!m(\overline{e})$.) After making an asynchronous call $x := e!m(\overline{e})$, the caller may proceed with its execution without blocking on the method reply. Here x is a future variable, and e and $\overline{e}$ are expressions. Thus, a future variable x refers to a return value which has yet to be computed. There are two operations on future variables, which control synchronization in Creol. First, the guard **`await`** $x?$ suspends the active process unless a return to the call associated with x has arrived. This blocks execution in the process, but allows other processes to be executed. Second, the return value is retrieved by the expression x.**`get`**, which blocks all execution in the object until the return value is available. The statement sequence $x := o!m(\overline{e})$; $v := x$.**`get`** encodes a *blocking call*, abbreviated $v := o.m(\overline{e})$ (often referred to as a synchronous call), whereas the statement sequence $x := o!m(\overline{e})$; **`await`** $x?$; $v := x$.**`get`** encodes a non-blocking, *preemptable call*, abbreviated **`await`** $v := o.m(\overline{e})$.

Time. We consider a discrete time model, comparable to a system clock which updates every n milliseconds. With this granularity of time, an object which executes a statement may, but need not observe that time has advanced. The expression **`now`** returns the present time, i.e., the global clock's value in the current state. Time values are totally ordered by the less-than operator; comparing two time values result in a Boolean value suitable for guards in **`await`** statements. In the model, the passage of time is implicitly observable via such **`await`** statements. From an object's local perspective, time can advance by either evaluating statements or by awaiting the passage of time. This model of time combined with Creol's blocking and non-blocking synchronization semantics, is powerful enough to express both process- and object-wide *progress* statements.

3 Deployment Components with Parametric Concurrency

Creol's object model is inherently concurrent, which means that for the actual deployment of a program it is necessary to map the logical concurrency of the model to physical computing resources. For this purpose, we introduce a notion of *deployment component* into the modelling language, which abstracts from the number and speed of the physical processors available to the component by a notion of concurrent resource. The granularity of the global time model defines the points in time when the executing system is observable. Concurrent resources may be consumed in parallel or in sequential order, which reflects the number of processors and their speeds relative to the granularity of the time intervals of the model. Thus, the logical concurrency model of the concurrent objects is controlled by their associated deployment component. A deployment component is parametric in the computational resources it offers to a group of dynamically created objects, which allows easy configuration of concurrent resources.

The execution inside a deployment component can be understood as follows. Let n be a natural number. Resources are modelled by a data type `Resource`

which extends the natural numbers with an "unlimited resource" ω, such that resource consumption is captured by subtraction, where $\omega - n = \omega$. Within a time slot, a deployment component with r concurrent resources is able to execute up to n execution steps in parallel, where $n \leq r$. Consider a deployment component D instantiated with r resources and let G be the set of concurrent objects which currently reside in the deployment component. Let $A \subseteq G$ be a subset of the concurrent objects on the component, such that objects in A are able to perform an execution step in their current state. Provided $|A| \leq r$, every object in A may consume a resource, leaving $r' = r - |A|$ resources available on the component. If there are remaining resources $(r' > 0)$, another cycle of execution steps may be performed for r' within the time slot by repeating this procedure.

In the modelling language, a deployment component D is declared by associating a name to a given quantity of concurrent resources r, capturing the actual processing capacity of D. For simplicity in this paper, a deployment component is a static entity, in contrast to class declarations which act as templates for dynamic generation of objects. A component is introduced by the syntax **`component`** $D(r)$, where D is the name of the component and r, of sort `Resource`, represents the concurrent resources of the component. The set of concurrent objects residing on the components, representing the logically concurrent activities, may grow dynamically. Thus, when objects are created, we require that they reside inside a deployment component. The syntax for object creation is extended with an optional clause to specify the targeted deployment component: **`new`** $C(\overline{e})$ **`in`** D. This expresses that C will reside in the component D. Objects generated by a parent object residing in a component D will also reside in D unless otherwise specified by an **`in`** clause. Thus the behavior of a Creol model which does not statically declare additional deployment components, can be captured by a main deployment component with ω resources.

4 Example: A Distributed Shopping Service

We consider a simple model of a web shop (see Fig. 2). Clients connect to the shop by calling the `getSession` method of an `Agent` object. An `Agent` hands out `Session` objects from a dynamically growing pool. Clients call the `order` method of their `Session` instance, which in turn calls the `makeOrder` method of a `Database` object that is shared across all sessions. After completing the order, the session object is added to the agent's pool again. This scenario models the architecture and control flow of a database-backed website, while abstracting from many details (load-balancing thread pools, data model, sessions spanning multiple requests, etc.), which can be added to the model should the need arise.

In the implementation of the `Database` class, an order takes a minimum amount of time, and should be completed within a maximum amount of time. The timing behavior of the database is configurable via the class parameters `min` and `max`. Line 8 implements the delay while processing the order, Line 9 calculates and returns the success status of the order (i.e., whether a timeout occurred). Note that a component with unlimited resources, will complete all

```
1  interface Agent { Session getSession(); Void free(Session session);}
2  interface Session { Bool order(); }
3  interface Database { Bool makeOrder(); }
4
5  class Database(Nat min, Nat max) implements Database {
6      Bool makeOrder () {
7          Time t:=now;
8          await now >= t + min;
9          return now <= t + max; }
10 }
11 class Agent(Database db, Set[Session] available) implements Agent{
12     Session getSession() {
13         if isempty(available) {
14             return new Session(this, db); }
15        else { session:=choose(available);
16               available:=remove(session,available);return session;}}
17     Void free(Session session){available:=add(available,session);}
18 }
19 class Session(Agent agent, Database db) implements Session {
20     Bool order() {return db.makeOrder(); agent.free(this); }
21 }
```

Fig. 2. A web shop model in Creol.

orders in the minimum amount of time, just as expected. In the `Agent` class, the attribute `available` stores a set of `Session` objects. (Creol has a datatype for sets, with operations `isempty` to check for the empty set, denoted `{}`, `choose` to select an element of a non-empty set, and `remove` and `add` to remove or add an element to a set.) When a customer requests a `Session`, the `Agent` takes a session from the available sessions if possible (Line 15), otherwise it creates a new session (Line 14). The method `free` inserts a session in the `available` sessions of the `Agent`, and is called by the session itself upon completion of an order. Section 6 shows how to run this example on a deployment component.

5 Operational Semantics

The semantics of Creol is defined in rewriting logic (RL) [20], and Creol models can be analyzed using the rewrite tool Maude [9]. In a rewrite theory (Σ, E, L, R), the signature Σ defines the ground terms, E defines equations between terms, L is a set of labels, and R a set of labeled rewrite rules. Rewrite rules apply to terms of given sorts, specified in (membership) equational logic (Σ, E). When modeling computational systems, different system components are typically modeled by terms of suitable sorts and the global state configuration is a set of these terms. RL extends algebraic specification techniques with transition rules: The dynamic behavior of a system is captured by rewrite rules supplementing the equations defining the term language. A *conditional rewrite rule* **crl** [*name*] : $t \to t'$ **if** c may be understood as a *local transition rule* allowing an instance of the pattern t to evolve into the corresponding instance of the pattern t', where the condition c is a conjunction of rewrites and equations that must hold for the main rule to apply (the *name* identifies the rule). When auxiliary functions are needed,

these can be defined in equational logic, and thus evaluated in between the state transitions [20]. In a *conditional equation* **ceq** $t = t'$ **if** c the condition c must similarly hold for the equation to apply. Both rewrite rules and equations may be unconditional, denoted by the keywords **rl** and **eq**, respectively. Given an initial configuration, Maude supports simulation and breadth-first search through reachable states and model checking of finite reachable states for desired properties. In this paper, Maude is used as an interpreter for Creol's operational semantics in order to simulate and test Creol models.

The States. Following Maude conventions runtime objects are represented by terms of the form $\langle o : C | \ \ldots, \ \texttt{Att:}\ x, \ldots \rangle$, where o is the identifier, C the class, and the object contains a set of attributes such that `Att` is the name and x the current value of an attribute. Variables are *slanted*, whereas constant parts of a term's syntax are in `typewriter` style. As before, $\overline{t}$ denotes a collection of terms t, either a list or a set depending on the context. Let `Emp` be the empty list and $\emptyset$ the empty set. In the rules below, all numbers are natural numbers (e.g., in counters and time) except resources which are of sort `Resource`.

A state *configuration* is a set which consists of a global clock, deployment components, objects, classes, invocation messages, and futures. The associative and commutative union operator on configurations is denoted by whitespace and the empty configuration by `none`. The *entire* configuration lives inside curly brackets; thus, in the term $\{\, cn \,\}$ the variable cn captures the entire configuration. The global *clock* is a term $\langle\, t\texttt{: Clock} \mid \texttt{limit:}\ l \,\rangle$ where t is the current time and l the time limit we consider in an execution of the semantics. A *deployment component* is a term $\langle\, dc\texttt{: Comp} \mid \texttt{Free:}\ r\texttt{,limit:}\ l \,\rangle$ where dc is the identifier of the component, r the (non-negative) number of available computing resources, and l the maximum number of resources which can be consumed before the clock advances.

An *object* is a term $\langle\, o\texttt{:}\ C \mid \texttt{Att:}\ \overline{a}\texttt{,Pr:}\ \{\overline{l} \mid \overline{s}\}\texttt{,PrQ:}\ \overline{w}\texttt{,Lcnt:}\ f\texttt{,Comp:}\ dc \,\rangle$ where o is the object's identifier and C its class, the object's state is given by the attribute mapping $\overline{a}$ (i.e., a single *binding* a associates a value with a declared variable), a *process* $\{\overline{l} \mid \overline{s}\}$ consists of a mapping $\overline{l}$ of local variable bindings and a list $\overline{s}$ of statements. The set $\overline{w}$ of (suspended) processes represents the process queue. The counter f is used to ensure that futures created by the object have unique identifiers, and dc is the deployment component associated with the object.

A *class* is a term $\langle\, C\texttt{: Class} \mid \texttt{Prm:}\ \overline{x}\texttt{,Att:}\ \overline{a}\texttt{,Mtds:}\ \overline{M}\texttt{,Ocnt:}\ y \,\rangle$, where C is the identifier, $\overline{x}$ the list of formal parameters, $\overline{a}$ maps declared attributes to initial (default) values, and $\overline{M}$ the set of method definitions of the form $\langle\, m\texttt{: Mtd} \mid \texttt{Prm:}\ \overline{x}\texttt{, Att:}\ \overline{l}\texttt{, Code:}\ \overline{s} \,\rangle$. Here, m is the method name, $\overline{x}$ formal parameter list, $\overline{l}$ the mapping of local variables to initial (default) values, and $\overline{s}$ a sequence of statements. The counter y will ensure that created objects get unique identifiers.

An *invocation message* is a term $\texttt{invoc}(o, n, m, \overline{d})$ where o is the callee, n the a future to which the call's result shall be returned, m the method name, and $\overline{d}$ lists the call's actual parameter values. A *future* is a term $\langle\, n\texttt{: Fut} \mid \texttt{Done:}\ b\texttt{,Value:}\ d \,\rangle$ where n is the identifier, b a Boolean flag indicating whether the future's reply value has been received, and d the reply value.

$$\textbf{rl}\ [\mathit{skip}]: \langle\, o : C \mid \mathtt{Pr}: \{\bar{l} \mid \mathtt{skip}; \bar{s}\},\ \mathtt{Comp}: dc \,\rangle\ \langle\, dc: \mathtt{Comp} \mid \mathtt{Free}: r \,\rangle$$
$$\longrightarrow \langle\, o : C \mid \mathtt{Pr}: \{\bar{l} \mid \bar{s}\},\ \mathtt{Comp}: dc \,\rangle\ \langle\, dc: \mathtt{Comp} \mid \mathtt{Free}: r - 1 \,\rangle\ .$$

$$\textbf{rl}\ [\mathit{assign}]: \langle\, o : C \mid \mathtt{Att}: \bar{a}, \mathtt{Pr}: \{\bar{l} \mid x := e; \bar{s}\}, \mathtt{Comp}: dc \,\rangle\ \langle\, t : \mathtt{Clock} \mid \rangle\ \langle\, dc : \mathtt{Comp} \mid \mathtt{Free}: r \,\rangle$$
$$\longrightarrow \textbf{if}\ x \in \mathrm{dom}(\bar{l})\ \textbf{then}\ \langle\, o : C \mid \mathtt{Att}: \bar{a}, \mathtt{Pr}: \{\bar{l}[x \mapsto [\![e]\!]^t_{(\bar{a} \circ \bar{l}), none}] \mid \bar{s}\}, \mathtt{Comp}: dc \,\rangle$$
$$\textbf{else}\ \langle\, o : C \mid \mathtt{Att}: \bar{a}[x \mapsto [\![e]\!]^t_{(\bar{a} \circ \bar{l}), none}], \mathtt{Pr}: \{\bar{l} \mid \bar{s}\}, \mathtt{Comp}: dc \,\rangle\ \textbf{fi}$$
$$\langle\, t: \mathtt{Clock} \mid \rangle\ \ \langle\, dc: \mathtt{Comp} \mid \mathtt{Free}: r - 1 \,\rangle\ .$$

$$\textbf{rl}\ [\mathit{if\text{-}then\text{-}else}]: \langle\, o : C \mid \mathtt{Att}: \bar{a}, \mathtt{Pr}: \{\bar{l} \mid \mathtt{if}\ e\ \mathtt{th}\ \bar{s}_1\ \mathtt{el}\ \bar{s}_2\ \mathtt{fi}; \bar{s}\} \,\rangle\ \langle\, t: \mathtt{Clock} \mid \rangle$$
$$\longrightarrow \textbf{if}\ [\![e]\!]^t_{(\bar{a} \circ \bar{l}), none}\ \textbf{then}\ \langle\, o : C \mid \mathtt{Att}: \bar{a},\ \mathtt{Pr}: \{\bar{l} \mid \bar{s}_1; \bar{s}\} \,\rangle$$
$$\textbf{else}\ \langle\, o : C \mid \mathtt{Att}: \bar{a},\ \mathtt{Pr}: \{\bar{l} \mid \bar{s}_2; \bar{s}\} \,\rangle\ \textbf{fi}\ \langle\, t: \mathtt{Clock} \mid \rangle\ \ .$$

$$\textbf{crl}\ [\mathit{return}]: \langle\, o : C \mid \mathtt{Att}: \bar{a},\ \mathtt{Pr}: \{\bar{l} \mid \mathtt{return}(e); \bar{s}\},\ \mathtt{Comp}: dc \,\rangle\ \langle\, t: \mathtt{Clock} \mid \rangle$$
$$\langle\, n: \mathtt{Fut} \mid \mathtt{Done}: \mathtt{false},\ \mathtt{Value}: \bot \,\rangle\ \langle\, dc: \mathtt{Comp} \mid \mathtt{Free}: r \,\rangle$$
$$\longrightarrow \langle\, o : C \mid \mathtt{Att}: \bar{a},\ \mathtt{Pr}: \{\bar{l} \mid \bar{s}\},\ \mathtt{Comp}: dc \,\rangle\ \langle\, n: \mathtt{Fut} \mid \mathtt{Done}: \mathtt{true},\ \mathtt{Value}: [\![e]\!]^t_{(\bar{a} \circ \bar{l}), none} \,\rangle$$
$$\langle\, t: \mathtt{Clock} \mid \rangle\ \ \langle\, dc: \mathtt{Comp} \mid \mathtt{Free}: r - 1 \,\rangle\ \textbf{if}\ n = \bar{l}(\mathtt{destiny})\ \ .$$

$$\textbf{rl}\ [\mathit{release}]: \langle\, o : C \mid \mathtt{Pr}: \{\bar{l} \mid \mathtt{release}; \bar{s}\}, \mathtt{PrQ}: \overline{w} \,\rangle$$
$$\longrightarrow \langle\, o : C \mid \mathtt{Pr}: \mathtt{idle}, \mathtt{PrQ}: \overline{w} \cup \{\{\bar{l} \mid \bar{s}\}\} \,\rangle\ .$$

$$\textbf{crl}\ [\mathit{await1}]: \{\langle\, o : C \mid \mathtt{Att}: \bar{a},\ \mathtt{Pr}: \{\bar{l} \mid \mathtt{await}\ e; \bar{s}\} \,\rangle\ \mathit{cn}\ \langle\, t: \mathtt{Clock} \mid \rangle\ \}$$
$$\longrightarrow \{\langle\, o : C \mid \mathtt{Att}: \bar{a},\ \mathtt{Pr}: \{\bar{l} \mid \bar{s}\} \,\rangle\ \mathit{cn}\ \langle\, t: \mathtt{Clock} \mid \rangle\ \}\ \textbf{if}\ [\![e]\!]^t_{(\bar{a} \circ \bar{l}), cn}\ \ .$$

$$\textbf{crl}\ [\mathit{await2}]: \{\langle\, o : C \mid \mathtt{Att}: \bar{a},\ \mathtt{Pr}: \{\bar{l} \mid \mathtt{await}\ e; \bar{s}\} \,\rangle\ \mathit{cn}\ \langle\, t: \mathtt{Clock} \mid \rangle\ \}$$
$$\longrightarrow \{\langle\, o : C \mid \mathtt{Att}: \bar{a},\ \mathtt{Pr}: \{\bar{l} \mid \mathtt{release}; \mathtt{await}\ e; \bar{s}\} \,\rangle\ \mathit{cn}\ \langle\, t: \mathtt{Clock} \mid \rangle\ \}\ \textbf{if}\ \neg [\![e]\!]^t_{(\bar{a} \circ \bar{l}), cn}\ \ .$$

$$\textbf{crl}\ [\mathit{activate}]: \{\langle\, o : C \mid \mathtt{Att}: \bar{a},\ \mathtt{Pr}: \mathtt{idle},\ \mathtt{PrQ}: \overline{w} \cup \{\{\bar{l} \mid \bar{s}\}\} \,\rangle\ \mathit{cn}\ \langle\, t: \mathtt{Clock} \mid \rangle\ \}$$
$$\longrightarrow \{\langle\, o : C \mid \mathtt{Att}: \bar{a}, \mathtt{Pr}: \{\bar{l} \mid \bar{s}\}, \mathtt{PrQ}: \overline{w} \,\rangle\ \mathit{cn}\ \langle\, t: \mathtt{Clock} \mid \rangle\ \}\ \textbf{if}\ \mathtt{ready}(\bar{s}, (\bar{a} \circ \bar{l}), cn, t)\ \ .$$

$$\textbf{crl}\ [\mathit{async\text{-}call}]: \langle\, o : C \mid \mathtt{Att}: \bar{a},\ \mathtt{Pr}: \{\bar{l} \mid x := e!m(\bar{e}); \bar{s}\},\ \mathtt{Lcnt}: f,\ \mathtt{Comp}: dc \,\rangle$$
$$\langle\, t: \mathtt{Clock} \mid \rangle\ \ \langle\, dc: \mathtt{Comp} \mid \mathtt{Free}: r \,\rangle$$
$$\longrightarrow \langle\, o : C \mid \mathtt{Att}: \bar{a}, \mathtt{Pr}: \{\bar{l}[x \mapsto n] \mid \bar{s}\}, \mathtt{Lcnt}: f + 1, \mathtt{Comp}: dc \,\rangle\ \langle\, dc: \mathtt{Comp} \mid \mathtt{Free}: r - 1 \,\rangle$$
$$\mathtt{invoc}([\![e]\!]^t_{(\bar{a} \circ \bar{l}), none}, n, m, [\![\bar{e}]\!]^t_{(\bar{a} \circ \bar{l}), none})\ \ \langle\, n: \mathtt{Fut} \mid \mathtt{Done}: \mathtt{false}, \mathtt{Value}: \bot \,\rangle\ \langle\, t: \mathtt{Clock} \mid \rangle$$
$$\textbf{if}\ n := \mathtt{label}(o, f) \wedge o \neq [\![e]\!]^t_{(\bar{a} \circ \bar{l}), none}\ \ .$$

$$\textbf{rl}\ [\mathit{bind\text{-}method}]:\ \mathtt{invoc}(o, n, m, \bar{d})\ \ \langle\, o : C \mid \mathtt{PrQ}: \overline{w} \,\rangle$$
$$\langle\, C: \mathtt{Class} \mid \mathtt{Mtds}: (\overline{M} \cup \{\langle\, m: \mathtt{Mtd} \mid \mathtt{Prm}: \overline{x}, \mathtt{Att}: \bar{l}, \mathtt{Code}: \bar{s} \,\rangle\}) \,\rangle$$
$$\longrightarrow \langle\, o : C \mid \mathtt{PrQ}: \overline{w} \cup \{\{\bar{l}[\mathtt{destiny} \mapsto n, \overline{x} \mapsto \bar{d}] \mid \bar{s}\}\} \,\rangle$$
$$\langle\, C : \mathtt{Class} \mid \mathtt{Mtds}: (\overline{M} \cup \{\langle\, m: \mathtt{Mtd} \mid \mathtt{Prm}: \overline{x}, \mathtt{Att}: \bar{l}, \mathtt{Code}: \bar{s} \,\rangle\}) \,\rangle\ \ .$$

$$\textbf{crl}\ [\mathit{receive\text{-}comp}]: \langle\, o : C \mid \mathtt{Att}: \bar{a}, \mathtt{Pr}: \{\bar{l} \mid x := e.\mathtt{get}; \bar{s}\}, \mathtt{Comp}: dc \,\rangle$$
$$\langle\, n : \mathtt{Fut} \mid \mathtt{Done}: \mathtt{true}, \mathtt{Value}: d \,\rangle\ \langle\, dc : \mathtt{Comp} \mid \mathtt{Free}: r \,\rangle$$
$$\longrightarrow \langle\, o : C \mid \mathtt{Att}: \bar{a}, \mathtt{Pr}: \{\bar{l} \mid x := d;\ \bar{s}\}, \mathtt{Comp}: dc \,\rangle\ \langle\, n: \mathtt{Fut} \mid \mathtt{Done}: \mathtt{true}, \mathtt{Value}: d \,\rangle$$
$$\langle\, dc: \mathtt{Comp} \mid \mathtt{Free}: r - 1 \,\rangle\ \textbf{if}\ \mathtt{n} = [\![e]\!]^t_{(\bar{a} \circ \bar{l}), none}\ \ .$$

$$\textbf{rl}\ [\mathit{object\text{-}creation}]: \langle\, o : C \mid \mathtt{Att}: \bar{a}, \mathtt{Pr}: \{\bar{l} \mid x := \mathtt{new}\ B(\bar{e}); \bar{s}\}, \mathtt{Comp}: dc \,\rangle$$
$$\langle\, t: \mathtt{Clock} \mid \rangle\ \langle\, dc: \mathtt{Comp} \mid \mathtt{Free}: r \,\rangle\ \langle\, B: \mathtt{Class} \mid \mathtt{Prm}: \overline{x}, \mathtt{Att}: \bar{a}_1,$$
$$\mathtt{Mtds}: \overline{M} \cup \{\langle\, \mathtt{init}: \mathtt{Mtd} \mid \mathtt{Prm}: \mathtt{Emp}, \mathtt{Att}: \emptyset, \mathtt{Code}: \bar{s}_1 \,\rangle\ \}, \mathtt{Ocnt}: g \,\rangle$$
$$\longrightarrow \langle\, o : C \mid \mathtt{Att}: \bar{a}, \mathtt{Pr}: \{\bar{l} \mid x := \mathtt{newId}(B, g); \bar{s}\}, \mathtt{Comp}: dc \,\rangle\ \langle\, B: \mathtt{Class} \mid \mathtt{Prm}: \overline{x}, \mathtt{Att}: \bar{a}_1,$$
$$\mathtt{Mtds}: \overline{M} \cup \{\langle\, \mathtt{init}: \mathtt{Mtd} \mid \mathtt{Prm}: \mathtt{Emp}, \mathtt{Att}: \emptyset, \mathtt{Code}: \bar{s}_1 \,\rangle\ \}, \mathtt{Ocnt}: g + 1 \,\rangle$$
$$\langle\, \mathtt{newId}(B, g) : B \mid \mathtt{Att}: \bar{a}_1[\mathtt{this} \mapsto \mathtt{newId}(B, g), \overline{x} \mapsto [\![\bar{e}]\!]^t_{\bar{a} \circ \bar{l}, none}],\ \mathtt{Pr}: \{\emptyset \mid \bar{s}_1\},$$
$$\mathtt{PrQ}: \emptyset,\ \mathtt{Lcnt}: 0,\ \mathtt{Comp}: dc \,\rangle\ \langle\, t: \mathtt{Clock} \mid \rangle\ \langle\, dc: \mathtt{Comp} \mid \mathtt{Free}: r - 1 \,\rangle\ .$$

Fig. 3. A timed rewriting logic semantics for Creol. In the rewrite rules, the variable r ranges over non-zero natural numbers to ensure that resource values are non-negative.

Evaluating Expressions. Given a substitution s, a time t, and a configuration cn, denote by $[\![e]\!]^t_{s,cn}$ a confluent and terminating reduction system which reduces an expression e to a data value. Let $[\![\textbf{now}]\!]^t_{s,cn} = t$. Let $[\![x?]\!]^t_{s,cn} = \texttt{true}$ if $[\![x]\!]^t_{s,cn} = n$ and there is a future $\langle n\texttt{: Fut | Done: true, Value: } d\rangle$ in cn (for some value d), otherwise $[\![x?]\!]^t_{s,cn} = \texttt{false}$. The remaining cases are fairly straightforward, looking up values for declared variables in s. The reduction of an expression always happens in the context of a given process, object state, and configuration. Thus, $s = \overline{a} \circ \overline{l}$, the composition of the object state $\overline{a}$ and the local variable bindings $\overline{l}$, the time t is the current global time, and the configuration cn is the current global configuration of the system (ignoring the object itself).

The Rules. The rewrite rules of the operational semantics transform state configurations into new configurations, and are given in Fig. 3. In the presentation of a rule, we follow the convention of Full Maude [9] and hide attributes in runtime objects unless they are needed for that specific rule.

Rule *skip* consumes a **`skip`** in the active process and a resource in its deployment component. Rule *assign* evaluates an expression e and assigns the value to a variable x in the local state $\overline{l}$ or in the attributes $\overline{a}$, as appropriate, consuming a resource in its deployment component. Rules *if-then-else* branches the execution depending on the value obtained by evaluating the expression e. (We omit the rule for while, which unfolds the while loop using an if-expression.)

Process suspension and activation. Rule *return* puts the return value into the future associated with the call (via the destiny-variable which refers to the appropriate future) and marks the flag done as true in that future. This operation consumes a resource. Rule *release* suspends the active process by placing it on the process queue. We denote by `idle` the idle active process. Rule *await1* consumes the await statement in the case where the guard evaluates to true in the current state of the object, rule *await2* adds a `release` statement to the process in order to suspend the process in the case where the guard evaluates to false. Rule *activate* selects a process from the process queue if the statement list of this process is *ready* to execute. A process is ready if it would not directly be suspended again or block the processor (the formal definition is given in [18]).

Communication and object creation. Rule *async-call* sends an invocation message to the callee with the actual method parameters and the identity of a future in which to place the method's return value. The caller creates the future associated with the call, with a unique identity $\texttt{label}(o, f)$ constructed from the caller's own identity o and a local counter f. The future's `Done` attribute is initially `false` and the return value is undefined (i.e., $\perp$). This operation consumes a resource. Rule *bind-method* consumes an invocation method and places the process corresponding to the method call in the process queue of the callee. Note that we use a reserved local variable `destiny` to store the identity of the future associated with the call. Rule *receive-comp* dereferences the future variable n in the case where the future's `Done` attribute is `true`. Note that if this attribute is `false` the reduction in this object is *blocked*. This operation consumes a resource. Finally, *object-creation* creates a new object with a unique identifier $\texttt{newId}(B, g)$ constructed from the class identifier B and a local counter

```
eq canAdv(cn',t) = true .              //cn' contains no objects or messages
eq canAdv (msg cn,t) = false .                 //messages are instantaneous
eq canAdv(⟨ o : C | ⟩ ⟨ dc: Comp | Free: 0 ⟩ cn,t)              //no more resources
   = canAdv(⟨ dc: Comp | Free: 0 ⟩ cn,t) .
eq canAdv(⟨ o : C | Pr:{l | n.get;s)}⟩        //o is blocked, value not available
   ⟨ n: Fut | Done: false ⟩ cn,t) = canAdv(⟨ n: Fut | Done: false ⟩ cn,t) .
ceq canAdv(⟨ o : C | Att: a,Pr: idle,PrQ: w ⟩ cn,t)          //no ready processes
   = canAdv(cn,t) if noneready(w,a,cn,t) .
eq canAdv(⟨ o : C | ⟩ cn, t) = false [owise] .

eq Adv(⟨ dc: Comp | Free: r,limit: max ⟩ cn)
   = ⟨ dc: Comp | Free: max,limit: max ⟩ Adv(cn) .
eq Adv(cn) = cn [owise] .

crl [progress]: {cn ⟨ t: Clock | limit: limit ⟩ }
⟶ {Adv(cn) ⟨ t + 1: Clock | limit: limit ⟩ } if canAdv(cn,t) ∧ t < limit .
```

Fig. 4. Advancing the clock. Here, *msg* denotes a message, r ranges over non-zero natural numbers (as before), and *cn'* ranges over message- and object-free configurations.

g. The object's state is generated from default values for state attributes, extended with the actual values for `this` and the class parameters. In order to instantiate the remaining attributes, the `init` method is loaded (we assume that this method reduces to `skip` if unspecified in the class definition, and that it asynchronously calls `run` if the latter is specified.) This operation consumes a resource. Note that the new object inherits the deployment component of its creator. The rule for object creation in a named deployment component differs from *object-creation* only on this point, and is omitted from the presentation.

Advancing time. We capture a *run-to-completion* semantics for concurrent execution within the resource bounds of deployment components: all objects must finish their actions as soon as possible if resources are available. In order to capture timed concurrent execution with an interleaving semantics, time cannot advance freely. Time advance is regulated by a predicate `canAdv`, ranging over configurations and time (see Fig. 4), which can be explained as follows:

- For simplicity, we here assume that invocation messages do not take time. Therefore, time may *not* advance when a message is on its way. (A timed model of communication may be obtained by introducing explicit delays in the model, associated with specific method calls, see Sect. 4.)
- If a deployment component has remaining resources and one of the component's objects o may perform an action, then time may *not* advance. There are three cases:
  1. the active process in o is blocked on a value that has become available,
  2. the active process in o is idle, but a suspended process of o can be activated. (The predicate `noneready` in the equation expresses that for all processes $\{\overline{l}|\overline{s}\} \in \overline{w}$, we have $\neg\mathtt{ready}(\overline{s}, \overline{a} \circ \overline{l}, cn, t)$.)
  3. the active process in o is not blocked.
- If a deployment component has run out of resources, none of its objects can proceed, and hence time can advance.

```
class SyncClient(Agent a,Nat c){
  Void run {
    Time t := now;
    Session s := a.getsession();
    Bool result := s.order();
    await now >= t + c; !run(); } }

class PeriodicClient(Agent a,Nat c){
  Void run {
    Time t := now;
    Session s := a.getsession();
    Fut(Bool) rc:= s!order();
    await now >= t + c;
    !run();
    await rc?; Bool r := rc.get; } }

component shop(10)
Void main() {
   Database db := new Database(5, 10) in shop;
   Agent a := new Agent(db, {}) in shop;
   PeriodicClient c := new PeriodicClient(a, 5); }
```

Fig. 5. Deployment environment and client models of the web shop example.

If there can be no activity in any object and no messages are in transit, then time may advance. Time advance is captured by the rewrite rule *progress* in Fig. 4, which updates the global clock. Once time has advanced, the deployment components get their resources refreshed for the next cycle of computation. This is done by an auxiliary function `Adv` defined in Fig. 4, which updates a configuration by resetting the free resources of each deployment component to the specified limit. Observe that for simplicity we here advance time with a single unit. It is of course straightforward to add an attribute *delta* which allows larger increments. However, this may lead to incompleteness for search in the timed models [21]. Furthermore, we add a *limit* to the global clock and only consider execution sequences up to this limit in time in order to ensure termination of model execution.

6 Simulating and Testing the Example

The web shop example of Section 4 is now extended by specifying a deployment component and an environment in order to obtain testing and simulation results. Figure 5 shows how the web shop may be deployed: a *deployment component* `shop` is declared with 10 resources available for objects executing inside `shop`. The initial system state is given by the `main` method, which creates a single database, with 5 and 10 as its minimum and maximum time for orders, an `Agent` instance, and (in this example) one client outside of `shop`. The classes `SyncClient` and `PeriodicClient` model customers of the shop. `PeriodicClient` initiates a session and periodically calls `order` every `c` time units; `SyncClient` sends an `order` `c` time units after the last call returned.

Figure 6 displays the results of two sets of simulation runs over 100 clock cycles. For synchronous clients, 5 to 25 clients and 10 to 50 resources on the `shop` deployment unit were used. From about 15 clients, the number of requests scales linearly with the resources, indicating that the system is running at full capacity even at 50 resources. With larger numbers of clients, the number of successful requests decreases since communication costs also increase with the client load. For the periodic case, the system gets overloaded much more quickly since clients will have several pending requests; hence, only up to ten periodic clients were

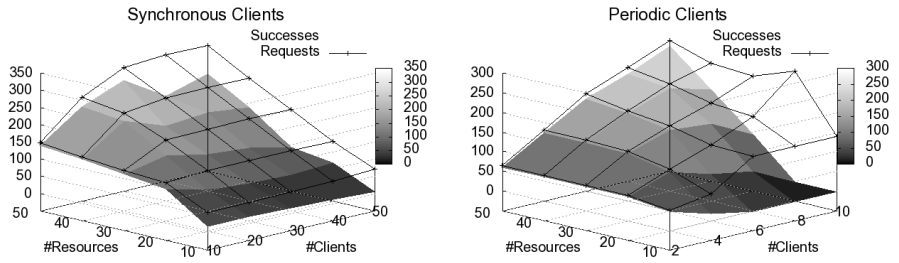


Fig. 6. Number of total and successful requests, depending on the number of clients and resources, for synchronous (left) and periodic (right) clients.

simulated. It can be seen that the system becomes completely unresponsive quickly when flooded with requests.

Testing Timed Observable Behavior. In software testing, a formal model can be used both for test case generation and as a test oracle to judge test outcomes [16]. For example, test case generation from formal models of communication protocols can ensure that all possible sequences of interactions specified by the protocol are actually exercised while testing a real system. Using formal models for testing is most widely used in functionality testing (as opposed to e.g. load testing, where stability and timing performance of the system under test is evaluated), but the approaches from that area are applicable to formally specifying and testing timing behavior of software systems as well [15].

In this paper, we model and investigate the effects of specific deployment component configurations on the timing behavior of timed software models. The *test purpose* in this scenario is to reach a conclusion on whether redeployment on a different configuration leads to an observable difference in timing behavior. Both *model* and *system under test* are Creol models of the same system, but running under different deployment configurations. In our example, the client object(s) model the expected usage scenario; results about test success or failure are relative to the expected usage. As *conformance relation* we use trace equivalence. This simple relation is sufficient since model and system under test have the same internal structure, hence we do not need to test for input enabledness, invalid responses etc. In our case, traces are sequences of communication events, i.e. method invocations and responses annotated with the time of occurrence, which are recorded on both the model and the system under test and then compared after the fact (off-line testing).

Running the model with five `SyncClients` (see Figure 5) but with an infinite number of resources in the deployment unit results in a trace $\langle 10, t\rangle$, $\langle 15, t\rangle$, $\langle 20, t\rangle$, ... (where each tuple contains $\langle$*response time, success*$\rangle$). Deploying with 50 resources results in the same trace, whereas running with 20 units results in a trace $\langle 12, t\rangle$, $\langle 17, t\rangle$, $\langle 22, t\rangle$, ... Assuming that model and system under test have identical untimed behavior, we conclude that a system without resource limits

and a deployment unit of 50 units behave equivalently under the assumed workload, whereas deploying with 20 units will lead to observably different behavior.

7 Related Work

The concurrency model provided by concurrent objects and Actor-based computation, in which software units with encapsulated processors communicate asynchronously, is increasingly attracting attention due to its intuitive and compositional nature (e.g., [2–4, 6, 10, 13, 26]). A distinguishing feature of Creol is the cooperative scheduling between asynchronously called methods [18], which allows active and reactive behavior to be combined within objects as well as compositional verification of partial correctness properties [3, 10]. Creol's model of cooperative scheduling has recently been generalized to concurrent object groups in Java [24] by restricting to a single activity within the group. In this paper, we further generalize the notion of concurrent object groups to a resource-constrained deployment component, where the allowed activity in a group per time interval is parametric in terms of concurrent resources, using a time model which simplifies the one presented in [19]. This allows us to abstractly model the effect of deploying concurrent object groups on deployment components with different amounts of processing capacity.

Techniques and methodologies for predictions or analysis of non-functional properties are based on either *measurement* and *modelling*. Measurement-based approaches apply to existing implementations, using dedicated profiling or tracing tools like, e.g., JMeter or LoadRunner. Model-based approaches allow abstraction from specific system intricacies, but depend on parameters provided by domain experts [11]. A survey of model-based performance analysis techniques is given in [5]. Formal systems using process algebra, Petri Nets, game theory, and timed automata (e.g., [7, 8, 12, 14]) have been applied in the embedded software domain, but also to the schedulability of tasks in concurrent objects [17]. That work complements ours as it does not consider resource restrictions on the concurrency model, but associates deadlines with method calls.

Work on modelling object-oriented systems with resource constraints is more scarce. Using the UML SPT profile for schedulability, performance and time, Petriu and Woodside [22] informally define the Core Scenario Model (CSM) to solve questions that arise in performance model building. CSM has a notion of resource context, which reflects the set of resources used by an operation. CSM aims to bridge the gap between UML specifications and techniques to generate performance models [5]. Closer to our work is M. Verhoef's extension of VDM++ for simulation of embedded real-time systems [25], in which architectures are explicitly modelled using CPUs and buses, and resources are bound to the CPUs. However, the underlying object models and operational semantics are very different. VDM++ is based on multithread concurrency, preemptive scheduling, and a strict separation between synchronous method calls and asynchronous signals, in contrast to our work with concurrent objects, cooperative scheduling, and caller decisions about synchronization. In contrast to our fairly

succinct rewriting logic semantics, the extension to VDM++ is embedded into VDM++ itself and defined in terms of 100 pages of VDM++ specifications [25].

8 Conclusions and Future Work

This paper reports initial work on resource requirements and timing for the deployment of object-oriented components. We extend Creol with a notion of deployment component which is parametric in its concurrent resources per time unit and formalize the operational semantics of object execution on deployment components in rewriting logic. Based on this formalization, we use the Maude rewrite engine to validate resource requirements that are needed to maintain the behavior of the concurrent objects when deployed with restricted resources.

The proposed model of deployment components is simple and flexible. The time granularity is defined implicitly by the use of time outs, allowing several statements to be executed in one time interval. In contrast, the execution cost of basic statements is fixed (abstracting from the evaluation of expressions). With a single resource, at most one basic statement can be executed inside a deployment component in a time interval. With multiple resources, all resources are used within the time interval if possible. This proposed resource model does not describe component scheduling policies, and abstracts from processor swapping costs. The model may be refined by associating deadlines to method calls and by defining explicit scheduling policies [17]. Furthermore, the model may be extended with the dynamic creation of deployment components as well as reconfiguration in terms of object mobility and resource adjustments, as well as stronger analysis methods such as, e.g., bisimulation techniques.

The abstract notion of resource proposed in this paper reflects computational limitations of concurrent or interleaved activity. Combined with the flexible time model, the proposed resource model can express interesting non-functional system properties, as illustrated by the example. Whereas most work on performance analysis assumes a fixed underlying architecture, we believe approaches such as the one presented in this paper address a need in software product lines which may vary in the underlying architecture of products.

References

1. E. Ábrahám-Mumm, F. S. de Boer, W.-P. de Roever, and M. Steffen. Verification for Java's reentrant multithreading concept. In *Proc. FOSSACS'02*, LNCS 2303, pages 5–20. Springer, Apr. 2002.
2. G. A. Agha. *ACTORS: A Model of Concurrent Computations in Distributed Systems.* The MIT Press, Cambridge, Mass., 1986.
3. W. Ahrendt and M. Dylla. A verification system for distributed objects with asynchronous method calls. In *Proc. Intl. Conf. on Formal Engineering Methods (ICFEM'09)*, LNCS 5885, pages 387–406. Springer, 2009.
4. J. Armstrong. *Programming Erlang: Software for a Concurrent World.* Pragmatic Bookshelf, 2007.

5. S. Balsamo, A. D. Marco, P. Inverardi, and M. Simeoni. Model-based performance prediction in software development: A survey. *IEEE Transactions on Software Engineering*, 30(5):295–310, 2004.
6. D. Caromel and L. Henrio. *A Theory of Distributed Object*. Springer, 2005.
7. A. Chakrabarti, L. de Alfaro, T. A. Henzinger, and M. Stoelinga. Resource interfaces. In *Proc. 3rd Intl. Conf. on Embedded Software (EMSOFT'03)*, LNCS 2855, pages 117–133. Springer, 2003.
8. X. Chen, H. Hsieh, and F. Balarin. Verification approach of metropolis design framework for embedded systems. *Intl. J. of Parallel Prog.*, 34(1):3–27, 2006.
9. M. Clavel *et al.* Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 285:187–243, Aug. 2002.
10. F. S. de Boer, D. Clarke, and E. B. Johnsen. A complete guide to the future. In *Proc. ESOP'07*, LNCS 4421, pages 316–330. Springer, Mar. 2007.
11. I. Epifani, C. Ghezzi, R. Mirandola, and G. Tamburrelli. Model evolution by run-time parameter adaptation. In *Proc. ICSE'09*, pages 111–121. IEEE, 2009.
12. E. Fersman, P. Krcál, P. Pettersson, and W. Yi. Task automata: Schedulability, decidability and undecidability. *Inf. and Comp.*, 205(8):1149–1172, 2007.
13. P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2–3):202–220, 2009.
14. M. Hennessy and J. Riely. Information flow vs. resource access in the asynchronous pi-calculus. *ACM TOPLAS*, 24(5):566–591, 2002.
15. A. Hessel, *et al.* Testing real-time systems using UPPAAL. In *Formal Methods and Testing*, LNCS 4949, pages 77–117. Springer, 2008.
16. R. M. Hierons, *et al.* Using formal specifications to support testing. *ACM Computing Surveys*, 41(2), 2009.
17. M. M. Jaghoori, F. S. de Boer, T. Chothia, and M. Sirjani. Schedulability of asynchronous real-time concurrent objects. *Journal of Logic and Algebraic Programming*, 78(5):402–416, 2009.
18. E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling*, 6(1):35–58, Mar. 2007.
19. E. B. Johnsen, O. Owe, J. Bjørk and M. Kyas. An Object-Oriented Component Model for Heterogeneous Nets. LNCS 5382, pages 257–279. Springer, 2008.
20. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.
21. P. C. Ölveczky and J. Meseguer. Abstraction and completeness for Real-Time Maude. In *Proc. 6th Intl. Workshop on Rewriting Logic and its Applications (WRLA'06)*, ENTCS 176: 5–27. Elsevier, 2007.
22. D. B. Petriu and C. M. Woodside. An intermediate metamodel with scenarios and resources for generating performance models from UML designs. *Software and System Modeling*, 6(2):163–184, 2007.
23. K. Pohl, G. Böckle, and F. Van Der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer, 2005.
24. J. Schäfer and A. Poetzsch-Heffter. JCoBox: Generalizing active objects to concurrent components. In *Proc. ECOOP 2010*. To appear in LNCS, Springer, 2010.
25. M. Verhoef, P. G. Larsen, and J. Hooman. Modeling and validating distributed embedded real-time systems with VDM++. In *Proc. 14th Intl. Symposium on Formal Methods (FM'06)*, LNCS 4085, pages 147–162. Springer, 2006.
26. A. Welc, S. Jagannathan, and A. Hosking. Safe futures for Java. In *Proc. OOPSLA'05*, pages 439–453. ACM Press, 2005.
27. S. M. Yacoub. Performance analysis of component-based applications. In *Proc. Software Product Lines (SPLC'02)*, LNCS 2379, pages 299–315. Springer, 2002.

JMLUnit: The Next Generation

Daniel M. Zimmerman and Rinkesh Nagmoti

Institute of Technology
University of Washington Tacoma
Tacoma, Washington 98402, USA
dmz@acm.org, rinkeshn@u.washington.edu

Abstract. Designing unit test suites for object-oriented systems is a painstaking, repetitive, and error-prone task, and significant research has been devoted to the automatic generation of test suites. One method for generating unit tests is to use formal class and method specifications as test oracles and automatically run them with developer-provided data values; for Java code with formal specifications written in the Java Modeling Language, this method is embodied in the JMLUnit tool. While JMLUnit can provide reasonable test coverage when used by a skilled developer, it suffers from several shortcomings including excessive memory utilization during testing and the need to manually write significant amounts of code to generate non-primitive test data objects. In this paper, we describe a successor to JMLUnit that can rapidly generate and execute millions of tests, using supplied test data of only primitive types, without consuming unreasonable amounts of memory. We also present results from initial test runs and comparisons with the original JMLUnit.

1 Introduction

Unit testing has been an important validation technique in software development processes for many years. In a typical unit testing process, a developer designs a set (or *suite*) of unit tests and runs them on the system under test. Each individual unit test is designed to demonstrate that some subset of the software (the *unit* being tested) performs appropriate actions and generates appropriate outputs given particular inputs and a particular starting state. The existence of a comprehensive unit test suite provides evidence for the stability, reliability, and security of the system, though it cannot guarantee the system's correctness.

Unfortunately, designing test suites is a painstaking, repetitive, and error-prone task, especially for large, complex software systems. Test developers can easily overlook critical situations that need testing or develop a test suite with poor *coverage*—that is, one that tests an insufficient fraction of a system's code or functionality. Moreover, the manual development and maintenance of test suites (regardless of quality) represents a significant portion of the development and maintenance costs for a complex software project.

To address both the coverage and cost issues, there has been significant research effort devoted to the automatic generation of high-coverage unit test suites

using techniques ranging from purely random test generation to the use of symbolic execution to find critical execution paths. While some of these techniques can provide reasonable test coverage at low cost, they all have various limitations and have seen little adoption by software developers.

This work focuses on improving one particular unit test generation technique that has been adopted by developers who use the Java Modeling Language (JML) to specify their software systems, namely the specification-based test generation embodied in the JMLUnit tool. After providing some background information about unit testing, JML, and the JMLUnit tool, we describe the limitations of JMLUnit for testing complex systems. We then explain how we improve upon the existing tool and present coverage results from tests generated by both the old and new tools to demonstrate our improvements. The goals of this work are to make the JMLUnit tool more effective and easier to use and, more importantly, to provide a platform upon which to conduct experiments with new test data generation techniques that are currently under development.

2 Background

2.1 Unit Testing

Unit testing is, essentially, the execution of individual components of a system (the *units*) in specific contexts to see whether they generate expected results. A single unit test has two main parts: the *test data*, which are the actual values for software entities such as method parameters that will be used to set up the state of the unit under test, and the *test oracle*, which is a piece of code that determines whether the behavior of the unit is "correct" when it is set up with the test data and executed. A system under test (hereafter, *SUT*) typically requires many unit tests, which are collectively called a *test suite*. The quality, or *coverage*, of a particular test suite can be measured in several ways [16]; for example, *code coverage* is the percentage of the executable code in the SUT that is actually executed when running the test suite.

The simplest way to create unit tests is to rely on human judgment: a developer sits down with a piece of software, decides what test data should be used and how to determine whether each test has passed or failed, and encodes this information manually. Despite the fact that many techniques for automated test data and test oracle generation have been developed over the last several years, most unit test generation is still done by hand, even in large systems. For example, the open-source Eclipse Development Platform[1] contains several thousand hand-written unit tests.

There are several ways to generate both test data and test oracles automatically. One such way, the focus of this work, is embodied in the JMLUnit tool (described in Section 2.3); we will briefly describe some others in Section 6.

[1] http://www.eclipse.org/

2.2 The Java Modeling Language

The Java Modeling Language (JML) [13] is a specification language for Java programs. It supports class and method contracts in a Design by Contract [14] style, as well as more sophisticated properties up to and including full mathematical models of program behavior. Several tools work with JML, including compilers, static checkers, test generators, and specification generators [6].

The *Common JML* tool suite is the original, and still most widely used, set of JML tools. It supports Java language versions up to 1.4 and includes a type checker (`jml`), a compiler (`jmlc`) that compiles JML annotations into runtime checks, a runtime assertion checker (`jmlrac`), a version of Javadoc (`jmldoc`) that generates documentation including JML specifications, and a unit testing framework (JMLUnit, described below).

Support for modern Java (1.5 and later) syntax in JML—including generic types, enhanced for loops, and annotations—is currently being developed in OpenJML,[2] based on the current OpenJDK[3] codebase, and JMLEclipse,[4] based on the Eclipse Development Platform.

2.3 JMLUnit

JMLUnit [7] is a unit testing framework for JML-annotated code. It takes advantage of JML runtime assertion checking (hereafter, *RAC*) to enable the automatic construction of test oracles that classify tests into three categories: *successful* (or *passed*), *unsuccessful* (or *failed*), and *meaningless*. Successful and unsuccessful tests are familiar concepts to developers experienced in unit testing. In the JMLUnit context, a successful test is one where a method is called and no RAC errors occur; this means that the method conforms to its specification with respect to that call. An unsuccessful test is one where a method is called with its precondition satisfied and a RAC error occurs during the method's execution; this means that the method does not conform to its specification, because once its precondition has been satisfied it must execute correctly without violating any assertions.

Meaningless tests, on the other hand, are not likely to be familiar to most unit testing practitioners. In the context of JMLUnit, a meaningless test is one where a method is called *without* its precondition satisfied, causing a RAC error before the method is executed. In JML (and other Design by Contract-based specification techniques), a method call is explicitly permitted to generate any result whatsoever when it is called without its precondition satisfied, ranging from an unchanged system state to a catastrophic system failure. Since any result of such a test must be acceptable by definition, there is no way for such a test to fail; a test that cannot fail gives no useful information and is therefore meaningless.

[2] http://jmlspecs.svn.sourceforge.net/viewvc/jmlspecs/OpenJML/
[3] http://openjdk.java.net/
[4] http://jmlspecs.svn.sourceforge.net/viewvc/jmlspecs/JmlEclipse/

Of course, test oracles generated from the JML specifications present in the SUT are necessarily limited by the scope of those specifications. Some JML specifications are not executable, so the runtime checker cannot catch all possible specification violations (though the range of violations it can catch is extensive). The more detailed and precise executable specifications exist for a method, the better the ability of the generated test oracles to discern the correctness of that method. Methods or classes with no executable specifications—that is, with only informal specifications or with formal specifications that cannot be checked at runtime—cannot be effectively tested using such test oracles. However, the problem of writing good executable class and method specifications, while extremely important, is beyond the scope of this work; we proceed under the assumption that good executable specifications are present in at least a reasonable fraction of any system we intend to test.

In addition to constructing a test oracle for every method in the SUT, JMLUnit also constructs a limited set of test data for each method. It uses a default set of values for each primitive type in the Java language as well as the `String` type, which it treats as a primitive type for testing purposes. For example, the default set of values for the `int` type is {`-1`, `0`, `1`} and the default set of values for the `String` type is {`null`, `""`} (`""` is the empty string). JMLUnit allows the developer to augment these default sets with additional values; the test code it generates has a clearly delineated "test data supply section" where the developer can specify data values to be used in addition to the defaults. Typically, JMLUnit generates two test classes (one containing the test oracles and one containing the test data) per class under test; however, there is also an option to relegate the test data for all classes under test to a single "test data generator" class. JMLUnit does no automatic test data generation for non-primitive types, relying solely on the developer to write the code that generates such test data.

The tests generated by JMLUnit are executable by JUnit,[5] one of the first and most widely used automated test execution frameworks for Java-based systems. They exhaustively use all combinations of the generated test data as parameters to each method under test. For example, consider method `m` in Figure 1, which takes one `int` parameter and one `String` parameter. JMLUnit has 3 default `int` values and 2 default `String` values, so `m` will be called 6 times during testing if only default values are used. If the default values are augmented with i additional `int` values and s additional `String` values, `m` will be called $(3 + i)(2 + s)$ times.

JMLUnit includes a custom JUnit test runner (`jml-junit`) that provides detailed reporting of test results and correctly handles meaningless tests; JUnit itself has no integrated concept of meaningless tests. The JUnit framework is also integrated into the Eclipse IDE and JMLUnit tests can be run directly from inside Eclipse, though doing so causes meaningless tests to be reported as passed tests and the test results to be reported with less detail.

[5] http://www.junit.org/

```
public class Exemplar {
  public Exemplar(String s0, String s1, String s2, String s3,
                  byte b, char c, Other o, Thing t) {
    // constructor body omitted
  }

  public int m(int one, String two) {
    // method body omitted
  }
}
```

Fig. 1. An exemplar of a Java class skeleton.

3 Shortcomings of JMLUnit

In the hands of a skilled developer, JMLUnit can generate tests with good coverage; however, it has several limitations that make it somewhat impractical to use for large, complex systems. One of these is that it does not attempt to automatically generate non-primitive test data, leaving that task entirely to the developer. This requires the developer to manually write methods that return specific test objects in response to specific requests. In its generated test classes, JMLUnit provides skeletons for these methods, which are intended to return specific test data objects indexed by integers.

Consider class `Exemplar` in Figure 1, which has a constructor with the same signature as one we used in our experiments. When JMLUnit generates tests for the `Exemplar` constructor, it creates a method to provide objects of class `Thing` for the last constructor parameter. The developer must fill in the body of that method so that, whenever JMLUnit requests the `Thing` with index n, the method returns whatever the developer has decided the nth `Thing` should be. In most cases, it is important that the test object be a fresh copy, because the order in which tests are run is not known a priori and reuse of test objects can cause test results to unintentionally depend on the order in which the tests are run. Similarly, it is important that the test objects be constructed deterministically, because otherwise the test results might vary across test runs even if nothing in the SUT has changed. This leads to an implementation style where data generation methods are large `switch` statements, with the developer writing code in each `case` of the `switch` statement to generate a single test object; in fact, the skeleton code generated by JMLUnit is exactly such a `switch` statement with a `default` case that generates no test data. Such code requires considerable developer effort both to write and to maintain.

In addition to requiring data generation methods as above, JMLUnit does not provide a reasonable way to specify distinct test data sets for distinct contexts. For the `Exemplar` above, JMLUnit generates and provides extension points for `String`, `char` and `byte` data sets, as well as providing extension points for the developer to generate data for `Other` and `Thing`; however, it only provides *one* such data set and extension point for each type. Thus, if the 4 `String` parame-

ters `s0` ... `s3` have significantly different requirements (e.g., `s0` must be parsable as a number while `s2` must be a capitalized last name with certain length restrictions), the developer must add test data to the single `String` data set that satisfies all these requirements. This results in many meaningless tests where numeric strings are used as names and vice-versa.

The most critical shortcoming of JMLUnit, however, is its memory utilization. Since it relies on JUnit as its execution engine, JMLUnit must construct an entire JUnit test suite in memory, including all the test data to be used, before a single test is run. As described above, JMLUnit exhaustively tests all combinations of the generated test data for each method under test; thus, a single method that takes multiple parameters can result in extremely large numbers of tests. For the `Exemplar` constructor, if the developer gives no additional values beyond the default sets for the primitive types and `String` and generates 2 test objects for each of the `Other` and `Thing` types, JMLUnit generates a total of 384 tests. However, in a more realistic scenario where the developer adds, e.g., 3 `char` values, 2 `byte` values, and 2 `String` values to the default sets and generates 4 test objects for each of the object types, JMLUnit generates 102,400 tests.

The combinatorial explosion caused by adding additional test values is not problematic in itself; each of those 102,400 tests would execute quite quickly on any modern machine. However, the fact that JMLUnit is forced to construct the entire test suite in memory before executing the tests is a serious problem, because it makes such test suites completely impractical to execute even on extremely capable hardware. We attempted to run such a test suite for a case study (described in Section 5) on our test machine, an Apple Xserve with two 3.0GHz quad-core Xeon processors and 18GB of memory; even allowing the Java virtual machine to use 16GB of heap space, we found that it exhausted available memory before giving the results of a single test.

4 JMLUnitNG: Improvements to JMLUnit

In order to test more complex systems with less developer intervention, we have created a new tool called *JMLUnitNG*. The new tool addresses the shortcomings described in the previous section while preserving most of the basic operating principles of the original JMLUnit.

4.1 Test Data Generation

The first shortcoming we address is the lack of non-primitive test data generation. To test `Exemplar`, we need test data of class `Thing`. `Thing` has at least one constructor, either the default no-argument constructor provided by Java in the absence of any constructor code or an explicit constructor that takes zero or more parameters.

If `Thing` has a default constructor, we can construct `Thing`s by using that default constructor. If `Thing` has explicit constructors, tests will be generated for each of them when we generate tests for class `Thing` itself; thus, construction of a

number of `Thing`s will necessarily be attempted as part of the testing process. We can use the `Thing` constructors and their test data to generate `Thing`s for use as test data in other contexts; if there are k tests generated for `Thing` constructors, that gives us at most k `Thing`s for testing other (non-constructor) methods of `Thing` and methods of other classes under test that take `Thing` parameters. We have at most k instances, rather than exactly k instances, because some of the constructor tests may be meaningless or may fail; such tests do not result in the creation of `Thing`s suitable for further testing.

We use Java reflection to generate these instances. Like JMLUnit, JMLUnitNG generates two classes—one containing test oracles and another containing test data—per class under test. In each test data class, JMLUnitNG creates an inner class that iterates over the instances that are successfully created during constructor tests. When we run JMLUnitNG on class `Exemplar`, which takes a `Thing` as a constructor parameter, JMLUnitNG inserts code into the test data class for `Exemplar` that uses Java reflection to search for the test data class for `Thing`. Later, when running the tests on `Exemplar`, JMLUnitNG can then find the test data class for `Thing` (if it exists on the classpath) and use it to obtain `Thing`s for testing. The developer can also directly specify `Thing`s, as in the original JMLUnit. If JMLUnitNG finds the test data class for `Thing` when the tests are run, and reflective test object generation is enabled, the generated `Thing`s are used in addition to the developer-specified `Thing`s; if not, only the developer-specified `Thing`s are used.

There are three main issues that arise when using reflection and constructor test cases to generate test data. The first issue is that it is possible to have cyclic dependencies; for example, a constructor (not necessarily the only constructor) of class `X` takes a parameter of class `Y` and a constructor (again, not necessarily the only one) of class `Y` takes a parameter of class `X`. This issue can be addressed in a straightforward, though perhaps not optimal, way: use cycle detection flags when instantiating objects, such that if an instance of `X` is requested when another instance of `X` is already in the process of being generated, the cycle is detected and stopped by providing a default (that is, generated by a default constructor) or developer-specified instance of `X` instead of dynamically constructing one from test data.

The second issue is that constructing test data reflectively does not take polymorphism into account. For example, given a method on a chessboard class that takes a `Piece` as a parameter, JMLUnitNG will attempt to generate `Piece` objects but will not attempt to generate, e.g., `Bishop` or `Knight` objects even if those classes extend `Piece` and have test data generators. This issue is difficult to address in the general case, such as when determining what types to generate for a method that takes an `Object` as a parameter. It can be addressed for certain classes, e.g., the Java Collections Framework, with simple test data generation rules (such as "generate an `ArrayList` where a `List` is required"). It can also be addressed for specific test scenarios by analyzing the inheritance relationships during test generation for only the classes under test; then, given a method with a parameter of type `Piece`, the subtypes of `Piece` that are explicitly under test

would be generated as test data for the method while the subtypes of `Piece` that are not under test would not be.

The third issue is that constructing test data reflectively does not account for interrelationships among classes under test. For example, `Exemplar` takes instances of `Other` and `Thing` as parameters; suppose it requires that the `Other` and `Thing` passed to it be related to each other in a specific way (such as sharing an identification number or other such attribute). In that case, reflectively constructing the `Other` and `Thing` to pass to the `Exemplar` constructor will not establish that relationship. However, this is an issue that is also encountered in developer-designed test data, where complicated setup operations may be necessary; therefore, we accept it as a limitation of the reflective test data generation approach.

We will show in Section 5 that, despite these issues, the use of reflection to generate test data objects from primitive types provides a significant improvement in automatic test coverage over the original JMLUnit.

4.2 Context-Dependent Test Data

The second shortcoming we address is the lack of context-dependent test data. As previously mentioned, JMLUnit provides default sets of data for primitive types, and extension points for the developer to specify additional data values for primitive types as well as data for non-primitive types. However, it only provides one such extension point per type, per class under test. Though the extension points do allow some flexibility—they take a parameter to designate how far nested a loop is in which a type is being used, for example—they do not allow a developer to specify specific sets of data to be used in specific contexts.

The main reason to specify sets of data for specific contexts is to help contain the combinatorial explosion of tests. If two of the `String` parameters to the `Exemplar` constructor are names, and the other two must be parsed as numbers or other reference codes, using the same set of `String`s for all 4 parameters will result in many meaningless tests. Specifying a set of `String`s for the names and another set of `String`s for the numbers/reference codes allows the developer to reduce the number of meaningless tests, and thus reduce the time it takes to run the test suite.

JMLUnitNG provides extension points for the developer to specify an individual set of test data for each parameter of each method under test. These extension points have data types and method signatures embedded in their names to uniquely associate each with a context; for example, method `Exemplar.m()`, declared as `int m(int one, String two)`, would have extension points with names like `int_one_m_int_String` (`int` data to be used for the `one` parameter of the method with signature `m(int, String)`) in the generated test class. For non-primitive types, these extension points invoke the reflective data generation code described earlier by default.

In addition to these extension points, JMLUnitNG also provides "global" extension points that allow the developer to add test data for all occurrences of a given type, as in the original JMLUnit; such global extension points have

names like `char_for_all`. The test data that is actually used at runtime for a given method parameter consists of the default test data set generated by JMLUnitNG, the global test data set associated with the data type, and the test data set associated specifically with that method parameter.

The addition of custom test data sets for individual method parameters allows developers to fine-tune their test suites and to easily integrate data from external test data generators into the system.

4.3 Iterators and Lazy Test Generation

The third shortcoming we address is JMLUnit's excessive memory utilization. There are two main causes of memory utilization when running automated tests: the need to generate all the tests in a test suite before executing the suite, and the recording of information about executed tests using in-memory data structures.

Since the tests generated by JMLUnit are extremely repetitive—each method is called many times, with parameter lists generated by taking the cross product of the test data sets for its parameter types—an ideal way to execute them would be to lazily generate the parameter lists as they are needed, rather than marshaling the parameter lists for all the individual method calls in memory as part of setting up the test suite. Unfortunately, the JUnit test execution engine does not support lazy parameter list generation; while it does have the ability to run parameterized tests, where a single test method is run repeatedly with multiple parameter lists, it requires the parameter lists to be stored in a two-dimensional array in memory, making it impossible to save memory by parameterizing the tests.

In order to enable lazy parameter list generation, we replaced JUnit with TestNG,[6] a Java-based test execution engine that is similar in concept to JUnit but has a different feature set. Like JUnit, TestNG supports the use of arrays as data sources for parameterized test methods; however, it also supports the use of *iterators* for this purpose. When it encounters a test method that uses an iterator as a data source, it executes the test method with parameter lists provided by the iterator until the iterator is empty. This allows us to implement lazy parameter list generation; by using iterators over primitive test data sets and the previously-discussed iterators that generate test objects of non-primitive types, we can create combined iterators that generate parameter lists for test methods while only keeping a single parameter list in memory at a time.

TestNG also supports another critical feature that helps to avoid excessive memory utilization: it allows the use of custom *test listeners* to record detailed information about executed tests, including the parameters used for testing and the exception, if any, that caused the test to fail or be skipped. Thus, instead of recording every test result in memory and processing that information at the end of a test suite's execution, as the previous version of JMLUnit does, we can record test results to disk in a streaming fashion as the tests are executed, with as much detail as we choose. As distributed, TestNG does record every test execution in

[6] http://www.testng.org/

memory—even if the default test listeners are disabled—in order to present a basic test report at the end of execution. However, with only minor changes to the TestNG source code, we were able to eliminate this in-memory recording while maintaining the ability to use other desirable TestNG features. With our modified version of TestNG, we can run test suites of essentially arbitrary size in a reasonable amount of memory, provided that there is sufficient disk space to log their results; we have successfully run hundreds of millions of tests using less than 1 GB of Java heap space.

The switch from JUnit to TestNG as a test execution environment therefore allows us to eliminate all the memory issues associated with JMLUnit. It also removes the need for a custom test runner that understands meaningless tests, because TestNG natively supports the concept of a *skipped* test; we simply record the meaningless tests as skipped, by intercepting the appropriate JML assertion errors and wrapping them in TestNG `SkipException`s. In addition, because TestNG supports functionality such as dependencies among tests and multiple forms of parallel testing, it provides a robust platform upon which to perform future automated test generation experiments.

5 Results

We have run our current version of JMLUnitNG on two different sets of Java classes. Both are relatively small; one is a small set of classes that implements chess pieces, and the other is a set of core classes from the *Kiezen op Afstand* (KOA) Internet-based remote voting system [12] constructed for the Dutch government by the Security of Software group at Radboud University Nijmegen.

The chess piece classes are largely testable in isolation, though they have a dependency on a `Team` class[7] that is used to indicate whether each piece is black or white and to enable the pieces to determine their legal directions of movement. The piece classes, which are named for the pieces whose movements they model, have methods that take no more than 3 parameters; the majority of their methods take fewer than 2 parameters. The piece classes tested here share a common interface (`Piece`) but do not take advantage of inheritance to factor out the common functionality of chess pieces into a shared parent class; thus, they all have similar structure.

The KOA classes, by contrast, are highly interrelated, with some taking instances of multiple others as constructor and method parameters. They also have a significantly greater number of method parameters on average, making the combinatorial explosion of test method calls more pronounced. The classes in the KOA system model components of the Dutch election system: `District` represents a voting district; `KiesKring` represents a *kieskring*, which is a region containing a collection of voting districts that are counted together for the purpose of proportional representation in the lower house of the Dutch parliament; `Candidate` stores information about a single candidate for office; `KiesLijst`

[7] This is a `class` in the chess code tested here, because we are working with a version of JML that only handles Java 1.4 constructs; it would be an `enum` in modern Java.

Class	Total Blocks	Covered Blocks Orig	Covered Blocks New	% Covered Orig	% Covered New
`Candidate`	197	0	0	0	0
`CandidateList`	659	0	0	0	0
`District`	98	13	74	13.3	75.5
`KiesKring`	299	29	207	9.7	69.2
`KiesLijst`	431	45	173	10.4	40.1
`VoteSet`	745	0	0	0	0
Total	2429	87	454	3.6	18.7

Table 1. Results for KOA classes with JMLUnit (Orig) and JMLUnitNG (New)

Class	Total Blocks	Covered Blocks Orig	Covered Blocks New	% Covered Orig	% Covered New
`Bishop`	367	0	247	0	67.3
`King`	390	0	270	0	69.2
`Knight`	362	0	242	0	66.9
`Pawn`	403	0	273	0	67.7
`Queen`	368	0	248	0	67.4
`Rook`	360	0	240	0	66.7
`Team`	10	1	8	9.1	80
Total	2260	1	1528	0	67.6

Table 2. Results for Chess classes with JMLUnit (Orig) and JMLUnitNG (New)

stores a list of candidates for a particular kieskring; and `CandidateList` stores information about the entire set of candidates, across all regions, for a single election.

We use EMMA,[8] a code coverage tool for Java, to measure the coverage of the tests generated by JMLUnit and JMLUnitNG. EMMA measures coverage in terms of *basic blocks*, which are sequences of bytecode instructions without any jumps or jump targets, rather than in terms of lines of source code. When a Java program is run under EMMA, it generates a report that lists all the classes loaded by the virtual machine, their methods, the number of basic blocks in each method, and the number of those blocks that were executed during the run.

Tables 1 and 2 show the block coverage provided by JMLUnit and JMLUnitNG, based on the data in the EMMA reports. Both sets of generated tests were run with default settings and without modifying the generated code. For the chess classes, 165 tests were automatically generated by JMLUnit and 7,108 were automatically generated by JMLUnitNG; for the KOA classes, 686 tests were automatically generated by JMLUnit and 3,017 were automatically generated by JMLUnitNG. The disparity—JMLUnitNG generates fewer tests for the KOA classes than for the chess classes, while JMLUnit does the opposite—is due to the fact that the constructors for the chess classes have significantly less

[8] http://emma.sourceforge.net

Class	Total Blocks	Covered Blocks	% Covered
`Candidate`	197	118	59.9
`CandidateList`	659	74	11.23
`District`	98	75	76.5
`KiesKring`	299	239	79.9
`KiesLijst`	431	266	61.7
`VoteSet`	745	167	22.4
Total	2429	939	38.7

Table 3. Results for KOA classes with JMLUnitNG and provided primitive data values

restrictive preconditions; while the default test data generate many possible parameter lists for constructing test objects, significantly fewer of those satisfy the constructor preconditions for the KOA tests than for the chess tests.

Since JMLUnit has no way to construct objects on which to call test methods, it fails to provide any test coverage other than for object constructors that take only primitive values (or accept `null`, which JMLUnit uses as a default). By contrast, JMLUnitNG covers significant fractions of the systems under test with no developer intervention.

Adding primitive and `String` data to the JMLUnit tests, for either set of classes, does not improve their coverage because JMLUnit still does not construct test objects. Adding primitive and `String` data to the JMLUnitNG chess tests does not improve the coverage significantly, because the default values for the primitive types are sufficient to test nearly everything that can be tested by JMLUnitNG; the polymorphism limitation mentioned in Section 4.1 prevents JMLUnitNG from automatically generating useful tests for the methods that handle capturing of pieces, which take parameters of type `Piece` (an interface shared by all the pieces), or for methods like `equals`. However, adding primitive and `String` data for the JMLUnitNG KOA tests has a significant impact, as the added data can be chosen to satisfy constructor preconditions that are not satisfied by the default data. Table 3 shows that block coverage more than doubled when a few carefully-selected primitive and `String` data values were added to the test data set; JMLUnitNG generated 1,351,351 tests for that run.

The test runs with default data ran in less than 10 seconds each; however, the JMLUnitNG test run with added data required approximately 3 hours to complete the 1,351,351 tests. We believe that the execution time can be dramatically improved through optimization of the reflective test data generation process, as well as by parallelizing the test executions. However, the completion of a million-test run is itself a dramatic improvement over the original JMLUnit tool; it would have exhausted the available 16 GB of Java heap space during the attempt and generated no results, while JMLUnitNG used less than 768 MB of heap space and reported that all the tests passed.

6 Related Work

As previously mentioned, considerable research has been (and continues to be) devoted to automatic test generation, most of it to the generation of test data rather than test oracles. We have insufficient space here to give even a complete overview of the current state of the art. We thus describe only the most closely related of the existing automated test generation systems.

Test oracles can be derived from a behavioral specification of the SUT, such as structured documentation [15], a formal model [9], or inline specification statements written in languages such as JML (as we have used here). Regardless of the type of behavioral specification, the basic idea is the same as we have employed: a test oracle is generated for each unit based on the specification of that unit; tests that are run with data that would violate the unit's requirements (preconditions, assumptions) are ignored, and a test is considered to pass if the unit's specification is not violated by the test execution.

Most automated test data generation falls into one or more of the following categories: *randomness-based*, where test data are generated randomly; *optimization-based*, where test data are optimized over multiple test runs based on coverage observations; *code-driven symbolic execution-based*, where symbolic execution [11] is used to compute test data that will exercise particular execution paths of the SUT; *specification- or model-based*, where constraint solving is used to generate test data based on a logical analysis of a specification or model of the SUT; and *verification-based*, where test cases are generated from attempts to formally verify the SUT. The latter two are most closely related to our approach.

Specification- and model-based test data generation methods, implemented in tools such as BZ-TT [1], JML-Testing-Tools [3] and UniTesK [4], use a logical analysis to compute partitions of the variables that fulfill the explicit case distinctions present in a formal specification or model of the SUT. Once the partitions have been computed, constraint solving or model finding is used to find concrete test data in each partition.

Verification-based test data generation (hereafter, *VBT*) is a recent development, based on the idea of generating test cases from attempts to verify systems with formal specifications [10]. VBT uses symbolic execution, with termination being enforced by a bound on the number of times loops and recursions are unwound; it differs from code-driven symbolic execution-based methods by generating test data from path condition formulae encountered at termination nodes in the symbolic execution tree. The VBT approach works well for code with simple branching statements (if...then, switch/case, constant-bounded loops) but not as well for code with generalized loops or recursion, because only a limited number of loop iterations and only a limited recursion depth can be dealt with. VBT has been implemented in the KeY verification system [2] and in Kiasan/KUnit [8]. A uniform framework for verification and testing has been formalized in HOL/Isabelle for a small target language [5].

JMLUnitNG is complementary to, not competitive with, the test generation methods and tools described above. While these methods and tools are relatively heavyweight, using automated theorem provers, constraint solvers and symbolic

execution engines, JMLUnitNG is extremely lightweight, using only the TestNG framework and Java's reflection mechanism. It is an instant replacement (and improvement) for developers who already use JMLUnit, and a one-step addition to the software build process for developers who use JML but have not yet adopted JMLUnit. It is easy to use, and the principles underlying its operation are easy for typical software developers and students to understand regardless of their level of experience with JML specifications and tools. For more advanced developers, it can also be used in conjunction with more heavyweight methods; rather than manually creating context-dependent test data sets for the JMLUnitNG test oracles, or relying solely on the default data sets and reflective data generation, developers can create their data sets using one or more other test data generation tools.

7 Conclusion

We have presented JMLUnitNG, a new unit test generation and execution framework inspired by the original JMLUnit tool and based on a modified version of the TestNG unit testing framework for Java. The current implementation has some shortcomings; as a proof of concept, it was directly evolved from the original JMLUnit and is based on the Common JML tool suite, so it cannot be used on code that contains modern Java constructs such as generic types. It does not contain solutions for two of the issues—cyclic dependencies and polymorphism—discussed in Section 4.1. When generating test data, it cannot reflectively construct instances of classes that have no public constructors, such as those that rely on factory methods. We have already designed and partially implemented a new version of the tool, independent of the Common JML tool suite, to address all these issues.

Despite these shortcomings, we consider our initial experiments with JMLUnitNG to be quite successful; the ability to generate and rapidly execute millions of tests and the automatic generation of test data of non-primitive types are substantial improvements over the functionality provided by the original JMLUnit, and the resulting benefits can be easily realized in any project that currently uses JMLUnit for specification-based testing. Moreover, JMLUnitNG provides significant new developer flexibility, including the ability to specify context-dependent test data. As such, it is not only an improvement over the original JMLUnit, but also a sound foundation for future test data generation experiments.

Acknowledgements

A portion of this work was funded by a 2008–09 award from the University of Washington Tacoma Chancellor's Fund for Research & Scholarship. In addition, the authors would like to thank Dr. Joseph R. Kiniry for his role in initial discussions about JMLUnitNG and his useful comments during both its development and the writing of this paper.

References

1. Ambert, F., Bouquet, F., Chemin, S., Guenaud, S., Legeard, B., Peureux, F., Vacelet, N.: BZ-TT: A tool-set for test generation from Z and B using constraint logic programming. In: Formal Approaches to Testing of Software (FATES) 2002, Workshop of CONCUR'02. Brno, Czech Republic (Aug 2002)
2. Beckert, B., Hähnle, R., Schmitt, P.H.: Verification of Object-Oriented Software: The KeY Approach. No. 4334 in Lecture Notes in Computer Science, Springer-Verlag (2007)
3. Bouquet, F., Dadeau, F., Legeard, B., Utting, M.: JML-Testing-Tools: A symbolic animator for JML specifications using CLP. In: 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). Edinburgh, U.K. (Apr 2005)
4. Bourdonov, I.B., Kossatchev, A., Kuliamin, V.V., Petrenko, A.: UniTesK test suite architecture. In: International Symposium of Formal Methods Europe (FME). Copenhagen, Denmark (Jul 2002)
5. Brucker, A.D., Wolff, B.: Interactive testing with HOL-TestGen. In: Fifth International Workshop on Formal Approaches to Testing of Software (FATES). Edinburgh, U.K. (Jul 2005)
6. Burdy, L., Cheon, Y., Cok, D., Ernst, M., Kiniry, J., Leavens, G.T., Leino, K., Poll, E.: An overview of JML tools and applications. International Journal on Software Tools for Technology Transfer (Feb 2005)
7. Cheon, Y., Leavens, G.T.: A simple and practical approach to unit testing: The JML and JUnit way. In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP) 2002. Lecture Notes in Computer Science, vol. 2374, pp. 231–255. Springer-Verlag (2002)
8. Deng, X., Robby, Hatcliff, J.: Kiasan/KUnit: Automatic test case generation and analysis feedback for open object-oriented systems. In: Testing: Academic and Industrial Conference Practice and Research Techniques (TAICPART). pp. 3–12. Windsor, UK (September 2007)
9. El-Far, I.K., Whittaker, J.A.: Model-based software testing. Encyclopedia on Software Engineering (2001)
10. Engel, C., Hähnle, R.: Generating unit tests from formal proofs. In: Tests and Proofs, First International Conference (TAP). Zürich, Switzerland (Feb 2007)
11. King, J.C.: Symbolic execution and program testing. Communications of the ACM 19(7), 385–394 (July 1976)
12. Kiniry, J., Morkan, A., Cochran, D., Fairmichael, F., Chalin, P., Oostdijk, M., Hubbers, E.: The KOA remote voting system: A summary of work to date. In: 2nd International Symposium on Trustworthy Global Computing (TGC). Lucca, Italy (2006)
13. Leavens, G.T., Cheon, Y., Clifton, C., Ruby, C., Cok, D.R.: How the design of JML accommodates both runtime assertion checking and formal verification. In: Proceedings of the International Symposium on Formal Methods for Components and Objects (FMCO) 2002. Lecture Notes in Computer Science, vol. 2852, pp. 262–284. Springer-Verlag (2003)
14. Meyer, B.: Object-Oriented Software Construction, 2nd Edition. Prentice-Hall (1988)
15. Peters, D.K., Parnas, D.L.: Using test oracles generated from program documentation. IEEE Transactions on Software Engineering 24(3), 161–173 (March 1998)
16. Zhu, H., Hall, P.A.V., May, J.H.R.: Software unit test coverage and adequacy. ACM Computing Surveys 29(4), 366–427 (Dec 1997)

Verification Based Test Case Generation for Scoped Memory in Safety-Critical Java

Gabriele Paganelli

Department of Computer Science and Engineering
Chalmers University of Technology, Gothenburg, Sweden
gabriele.paganelli@chalmers.se

Abstract. Applications in Safety Critical Java can make use of scoped memory areas. Objects residing in different scoped memory areas cannot refer to each other arbitrarily. A unit's specification has therefore an added dimension, i.e. the description of the relation that holds among the caller object and the reference type arguments via a set of constraints. A method is proposed to extend verification-based test generation for applications following certain programming guidelines. The information resulting from running the procedure on the application can then be used by a verification-based test generation tool to generate test cases for the selected units. The analysis provides a criterion to select a subset of all the possible configurations satisfying the precondition of the unit under test that are likely to appear in the application it is used in.
Keywords: Safety-Critical Java, Verification-based testing, Unit testing, Test case generation, Constraint solving on finite domains

1 Introduction

This paper proposes a method to help derive a reasonable set of initial states for unit tests in a program that uses scoped memory areas, one of the features of the Real-Time Specification for Java (RTSJ) [2]. The motivation for this is essentially to filter out a potentially infinite set of test cases, by analyizing the entire program via static analysis.

The main issue with memory management in RTSJ is the concept of scope. Since garbage collection is a dangerous source of unpredictability (the garbage collector can preempt a thread with hard real-time deadlines causing it to miss them), in RTSJ a set of classes representing memory areas has been provided to programmers in order to manage and allocate memory chunks (scopes, or scoped memory areas) that then are reclaimed by the JVM, and threads executing in these areas are never interrupted by garbage collection. Objects can be allocated in different parts of memory. The relations that different objects can have according to their position in memory affects the testing effort: in principle all possible configurations of allocation should be taken into account. The memory model from the thread's point of view is a stack holding the scopes entered but not yet left. Every thread maintains a stack that keeps track of its memory usage. The model encodes the scoped behavior of the memory allocation: as a scope is entered, it is stacked in the thread's scope stack; in order to leave a certain scope

s, all the scopes above it (*inner scopes*) in the stack (that were entered after entering s) must have been left by the thread (which does not necessarily mean that the memory has been reclaimed, though). When a thread is executing in a scoped memory area s, all object allocations happen in s; so the created objects *reside* in s. A scoped area keeps track of how many threads are executing in it; when the counter goes to zero, the area is deallocated by the compliant JVM. The code to be executed in a scoped memory area s must be contained in the `run()` method of an object implementing the `Runnable` interface, passed to the object `o_s` representing s and executed by invoking `o_s.enter()`. The access to the memory cannot happen in an arbitrary way to prevent dangling references, e.g. an object in a scope *out* cannot reference an object in a scope *inn* that is more inner than *out*, because its life span is shorter. This introduces a new dimension in the specification of the behaviour of a unit which reflects itself on the way a unit should be tested. The following snippet shows a typical usage of scoped memory:

```
String outermost;
ScopedMemory outerScoped = new LTMemory(1000);
outerScoped.enter(new Runnable(){
  public void run(){
    String s = new String("String residing in outerScoped area.");
    System.out.println(s);
  }
});
```

An `LTMemory` object represents a scoped memory area with allocation time linear in its size (that is 1000 bytes in this case). In the above code, the object `s` resides in the memory area represented by `outerScoped`.

Let `u` be a method of class `D` along with its JML specification[1] and signature (Fig.1): The specification tells the client of `u` that if `a` resides in a more outer

```
Class D{
  /*@
    @ public normal_behavior
    @ requires \inOuterScope(a,b);
    @ ensures \result == 3;
    @*/
  public int u(A a,B b){...}
}
```

Fig. 1. Example specification and signature of method `u`.

scope than `b` when `u` is invoked, then the return value of `u` will be `3`. If it is not

[1] Note that the `\inOuterScope` construct is not standard JML. See later in the paper, or [9].

the case, nothing can be said about the state after the unit returns. In the least constrained situation, where one is using RTSJ without any restriction, there are plenty of configurations that would be candidates for testing this unit. The only constraint is that the object `a` must be in a more outer scope than the one in which `b` is allocated. `this` is not constrained at all, or it can be assumed to reside in an outer scope than the active one by default. If for instance the depth of the scope stack is constrained to be 3, this allows at least 18 test cases as shown in Fig.2. The following snippet is a possible corresponding test case for the configuration shown in Fig.3.

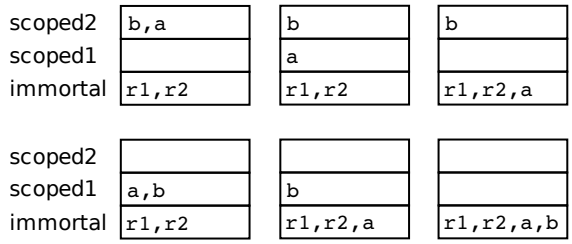


Fig. 2. Resulting test configurations for stacks of length 3. Note that here the `Runnable` objects that run inside the two scoped memory areas are assumed to be allocated in immortal memory, and it is assumed that the invocation happens always in the topmost scope. `this` can be allocated in all three memory areas, therefore giving $6 \times 3 = 18$ test contexts.

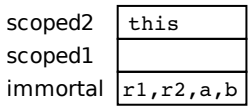


Fig. 3. Test case for the proposed configuration in JUnit format.

```
Runnable r1,r2;
LTMemory scoped1,scoped2;
A a;
B b;
@Begin
public void fixture(){
  scoped1 = new LTMemory(1000);
  scoped2 = new LTMemory(1000);
  r1 = new Runner1();
  r2 = new Runner2();
  B b = new B(...);
  A a = new A(...);
```

```
}

class Runner1 implements Runnable{
  public void run(){
    scoped2.enter(r2);
  }
}

class Runner2 implements Runnable{
  public void run(){
    int ret = d.u(a,b);
    Assert.assertEquals(ret,3);
  }
}

@Test
public void test(){
  scoped1.enter(r2);
}
```

This situation, besides being impractical, is also rather unrealistic. In the case in which one would derive test cases picking some of the many possible, one should come out with a set of criteria that would have little meaning if not related with the context in which the unit is going to work. Very often in fact real-time applications, and especially safety-critical applications, are strongly constrained by coding guidelines [7,16]. On one side this can reduce the expressivity of the language along with the portability and extendibility of the application. On the other side it ensures that the program can be easily understood by developers. This also frames the testing attempt, providing criteria to select the possible test inputs.

The rest of the paper is organized as follows. Section 2 gives the needed background to understand the rest of this work. Section 3 deals with the restrictions on coding that are assumed. Section 4 illustrates how the proposed method works, providing an example to show how effective it can be for selecting test cases. Conclusions are in Section 5.

2 Background

This section provides the needed background knowledge. It does not have any ambition of completeness on the topics presented.

2.1 Real-Time Java Scoped Memory Model

The Real-Time Java specification (RTSJ) [2] (started with JSR-1) represents the effort to bring Java in the real-time world. It does not extend the syntax of the language, but provides a set of new classes. Safety-Critical Java (SCJ) (JSR-302) is a profile for safety-critical applications that allows certification for

the DO-178B Level A standard [20]. The specification for the SCJ profile is still under definition, but several proposals exist [6,21] and they all agree on the fact that garbage collection should be avoided. For this paper's sake, the only important part is the way memory is managed. Normal heap memory, being subject to garbage collection, is not used in a safety-critical environment. There are two other kinds of memory areas: *immortal* and *scoped*. They all have a corresponding class representing them in the specification. Both are not garbage collected. Class `ImmortalMemoryArea` has a single object, representing immortal memory. Immortal memory is never reclaimed, therefore introducing the risk of memory leaks. Scoped memory areas are represented by `ScopedMemory` and its subclasses. A scoped memory area can be allocated at runtime, and reclaimed by a compliant JVM when the last thread executing in it leaves it, after running any finalization code associated with the allocated objects within it. This new feature allows the programmer to manage in a flexible way the used memory. A thread maintains a cactus stack (i.e. a stack with branches, or a tree that grows or shrink by adding or deleting leaves) of the memory areas it entered and not left yet. There are two ways to enter a memory area; with the `enter()` method or the `executeInArea()` method. The former stacks a new scope on the area in which the invoking thread is executing making the new scope the active one; the latter allows to move the active scope down the current scope stack.

As scoped memory areas can be reclaimed, there is the risk of dangling references. This can happen if a reference to an object residing in an inner scope in the memory stack is stored in a more outer scope, as for instance could happen in the following example:

```
String outermost;
ScopedMemory outerScoped = new LTMemory(1000);
outerScoped.enter(new Runnable(){
  public void run(){
    String s = new String("String residing in outerScoped area.");
    outermost = s; // throws IllegalScopeException
  }
});
```

or if a memory area is entered twice from the same thread while it is still active in its scope stack, which means that the single parent rule[2] has been broken [23]. A strict complying JVM should perform runtime checks each time an object reference is stored. This introduces an overhead. Previous works presented programming models and tools to process programs written according to such models to ensure that no scoped memory related exception will be thrown, therefore allowing to turn off runtime checks [6,11,17,18,22,24].

[2] The single parent rule states that any scoped memory area s must have a unique parent, where the parent is either:

- if there are scoped memory areas below s in the stack, the closest below s,or
- if there are no scoped memory areas below s, the memory area termed *primordial*.

2.2 JML

JML is a behavioral interface specification language that can be used to specify the behavior of Java modules [13]. It can be embedded in Java source files to annotate Java code, or in separate `.jml` files in the form of formatted comments. It allows to define the behavior in a precondition-postcondition fashion, also with class invariants and other features like ghost and model fields to support automated verification. The following snippet is a simple example of JML usage.

```
/*@
  @ public normal_behavior
  @ requires b>=0;
  @ ensures \result == a+b;
  @*/
public int sum(int a, int b){
  while(b>0){
    a++;
    b--;
  }
  return a;
}
```

The implementor has to fulfill the property that if the precondition (`requires`) is met by a client when it invokes the unit, then the postcondition (`ensures`) must hold when the method returns. On the client's side, the specification is a guarantee that if the precondition holds when the method is invoked, then when the method returns the postcondition will be true. Otherwise, nothing can be said about the behavior of the unit. `\result` here is a JML built-in expression referring to the return value of a method (if it is not `void`). For a slightly deeper, but still very easy introduction to the topic, refer to [14,15]. For a deeper understanding, refer to the language specification [13].

2.3 JML Extensions for Scoped Memory

In [9] a formalization of the stacking relation between scoped memory areas has been proposed. In the same work, an extension to allow JML to predicate about this relation is also developed. Two constructs of the latter are presented:

- `\inOuterScope(i,j)` indicates that the object `i` evaluates to is stored in the same or in a more outer scope than the object `j` evaluates to in the memory stack of the thread invoking the method.
- `\inImmortalMemory(i)` indicates that object `i` evaluates to is stored in immortal memory.

In the proposed extension, there is also the pointer `\currentMemoryArea`. This expression evaluates to the currently active memory area in which the state is evaluated. It will not be used in the following, but some remarks are given in Section 5.

2.4 Testing, Verification-Based Testing and KeYVBT

In this paper unit testing is taken into account, which is the test of single Java methods. A popular tool to execute and develop unit tests is JUnit [3].

Verification-based testing has been described in [8,10]. It is a testing approach that joins white box testing (by means of symbolic execution) and black box testing (because it uses the specification of a method). KeYVBT [5] is a verification-based testing tool that allows to automatically generate test cases for a subset of the Java language that includes the Java Card specification [1]. In it, the JML-annotated code is symbolically executed within the KeY system [4] in order to get the feasible execution paths and the associated path condition. Every feasible path then corresponds to a test case, generated in JUnit format.

3 Structure of the Program Under Test

As development of hard real-time programs poses stringent requirements, guidelines were proposed [17] to make SCJ applications safer, more reliable and easier to maintain. Program-wide analysis approaches to certify the properties of a program exist [11,22]. The JamaicaVM [7] manual also suggests several coding practices for hard real-time programming. In the following a series of assumptions are made. The application is a two-phase process: initialization and mission phase, the latter having to respect hard real-time constraints. Mission phase runs only in scoped memory areas of type `LTMemory`, whereas initialization runs in `ImmortalMemoryArea`. The use of just immortal and scoped memory areas ensures that the computation will never be interrupted by the garbage collector's activity. All the `LTMemory` objects are created during initialization. Dynamic loading of classes is not allowed in mission phase, so any application should load all classes needed during initialization. This limitations are consistent with the ones listed in [7,11,17], where the rationale behind them is discussed. Additional simplifications are added. Only single threaded applications with no invocations to `executeInArea()` are allowed, which means that the execution happens always in the memory scope at the top of the memory stack. Furthermore, the number of `LTMemory` used is known at compile time, and there is a one-to-one associations from `LTMemory` objects to `Runnable` classes (defined for simplicity in separate files).

In the following, $P = \{\rho_1, ..., \rho_n\}$ and $M = \{\mu_1, ..., \mu_n\}$ will denote the set of classes implementing `Runnable` and the instances of `LTMemory` in an application, respectively. The instances of ρ_i will run in μ_i, for all $i \in \{1, ..., n\}$. Imm is the immortal memory. Let also $M^* = M \cup \{Imm\}$ be the set of all memory areas used in the application.

Executions of such programs can therefore be represented, from the scoped memory point of view, as a sequence over time of linear stacks growing and shrinking continuously (in the sense that the previous stack is the next stack with or without a memory area stacked on top) in which the topmost memory is the active one (the one in which the thread is executing). Figure 4 shows a

possible stack development. Invoking `enter()` on a scoped memory area object `mem` representing memory area μ stacks the memory area; returning from that invocation unstacks it.

In the following, given a Java method u, the expression σ_b^u will indicate the specification of such a method. Furthermore, the expression $\tilde{\sigma}_b^u$ will indicate the subset of the specification dealing with memory constraints. The index b indicates multiple behavior specifications. Such a fragment of specification $\tilde{\sigma}_b^u$ shall contain only sequences of constraints

```
\inOuterScope(i,j)
\inImmortalMemory(i)
```

where `i`, `j` are reference type JML expressions, referring to formal parameters in the argument list or `this` (if the unit is not a static Java method) or more generally to any visible field mentioned in the specification (including class invariants).

The subset of the specification s that corresponds to the precondition (postcondition) will be denoted $Pre(\sigma_b^u)$ ($Post(\sigma_b^u)$). The structure of such a constrained program is therefore the following: the finite sets P, M^* represent respectively the `Runnable` implementations and the memory areas. A set of classes $\Gamma \supseteq P$ contains the classes used by the program. In the initialization phase, the program runs in immortal memory and instantiates the static fields, executes the static blocks, and performs other possible object creations: all the `LTMemory` objects in set M are created, and at least the `Runnable`(s) that will be entered to start mission phase. It also must force any dynamic loading of classes. Every stack during mission phase is rooted in immortal memory.

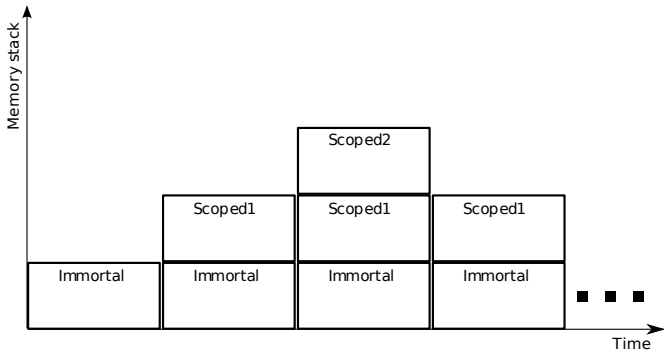


Fig. 4. Evolution over time of the described restricted program. The time ticks correspond to invoking (or returning from) the `enter()` method on the stacked memory area.

4 The Method

In the following it is assumed that the program has been already proved to not throw any runtime memory-related errors. Static analysis, and more precisely data-flow analysis, has been already used [22] to investigate the properties of SCJ programs, especially for verifying the absence of memory-related runtime errors. The aim is to produce initial states for a set of test cases. What is proposed here is a way to find mappings ("contexts") from objects to memory areas in all the memory stacks in which the method under test is invoked (the actual instantiation of these objects is not addressed here). These mappings, for the restriction imposed to the programs, can be found by static analysis. The problem is basically a constraint satisfaction problem: given certain constraints ($Pre(\tilde{\sigma}_b^u)$) the goal is to find all satisfiable assignments (that correspond to valid initial states) of the objects expressed in the specification and signature to memory areas in a certain memory stack. However, the method generates contexts that might not occur in the program, because of the coarse analysis; however, all contexts do respect the precondition and therefore can be used to build valid test cases.

4.1 Static Analysis

Let U be the set of Java methods u to test. For simplicity, three data structures are defined to name these three different views:

1. A tree T whose root is labeled with Imm and its subtrees have their roots labeled with the memory scopes entered from the parent context[3].
2. A table $\Theta_A : \Gamma \longmapsto \wp(M^*)$ that holds, for each class $\gamma \in \Gamma$, the set of memory areas in which instances of γ are created.
3. A table $\Theta_I : U \longmapsto \wp(M^*)$ that holds, for each Java method u to test, the set of memory areas in which it is invoked.

4.1.1 Example. A possible triple $\langle T, \Theta_A, \Theta_I \rangle$ obtained by static analysis, when $\Gamma = \{\texttt{R1}, \texttt{R2}, \texttt{R3}, \texttt{A}, \texttt{B}, \texttt{D}\}$, $P = \{\texttt{R1}, \texttt{R2}, \texttt{R3}\}$, $M = \{\texttt{M1}, \texttt{M2}, \texttt{M3}\}$:

$$
\begin{aligned}
T &= (\texttt{Imm}, \{(\texttt{M1}, \{(\texttt{M2}, \{\ (\texttt{M3}, \emptyset)\})\})\}) \\
A &= \{ \\
&\quad \texttt{R1} \rightarrow \{\texttt{Imm}\}, \\
&\quad \texttt{R2} \rightarrow \{\texttt{Imm}\}, \\
&\quad \texttt{R3} \rightarrow \{\texttt{Imm}\}, \\
&\quad \texttt{A} \rightarrow \{\texttt{M1, M3}\}, \\
&\quad \texttt{B} \rightarrow \{\texttt{M2, M3}\} \\
&\quad \texttt{D} \rightarrow \{\texttt{M2}\}, \\
&\quad \} \\
I &= \{\texttt{u} \rightarrow \{\texttt{M2, M3}\}\}
\end{aligned}
$$

[3] This induces a partial order among the memory areas, if the program has been already proved to not throw any runtime memory-related errors.

where `Imm` is the immortal memory, and a tree node is represented as the pair $(label, set\_of\_children)$.

4.2 Generation of the Contexts

Let $u \in U$ a unit to test. The possible contexts in which the unit could be run are then inferred from the triple $\langle T, \Theta_A, \Theta_I \rangle$ and $Pre(\tilde{\sigma}_b^u)$. The memory areas in which u is invoked are given by the set $\Theta_I(u)$. The occurrences of these memory areas are then searched in T. Note that this might happen in more than one place in T. The occurrences are described as a set of paths $\hat{s}_k = \langle Imm, \mu_h, ..., \mu_l \rangle$ each representing a memory stack. Let $S = \{\hat{s}_1, \hat{s}_2, ..., \hat{s}_j\}$ denote this set. Let s_k be the (unordered) set containing all elements appearing in a sequence $\hat{s}_k$. Note that for the finiteness of M^*, and the restrictions that prevent entering scopes more than once (see Subsec. 2.1), all the above mentioned sets and sequences are finite. $Pre(\tilde{\sigma}_b^u)$ contains the constraints that the allocation of object must fulfill.

4.2.1 Translation to Constraint Solving over Finite Domains. A constraint satisfaction problem (CSP) is defined as a triple $\mathcal{P} = \langle X, D, \mathcal{C} \rangle$ where X is an n-tuple of variables $X = \langle x_1, x_2, ..., x_n \rangle$ and their domains are described by the tuple $D = \langle D_1, D_2, ..., D_n \rangle$ such that $x_i \in D_i$. The constraints are given by $\mathcal{C} = \langle C_1, C_2, ..., C_t \rangle$, where each C_i is a pair $\langle R_{S_i}, S_i \rangle$. S_i is a set containing the variables constrained by C_i, and R_{S_i} is a relation among these variables that defines the valid combinations of values [19].

Given a method u, $Pre(\tilde{\sigma}_b^u)$ and $\langle T, \Theta_A, \Theta_I \rangle$, one can see the problem of satisfying $Pre(\tilde{\sigma}_b^u)$ as a constraint solving over finite domains. Considering every path $\hat{s}_i$ in T that describe where u is invoked, define for every

- reference-type formal parameter in u's signature
- reference-type visible field mentioned in the specification (including class invariants)
- `Runnable` ρ_j associated with memory scope μ_j occurring on $\hat{s}_i$

a variable taking values over a domain $D_r \subseteq M^*$ in which the corresponding object can be created. Note that this information is related with the information held in table Θ_A. Let $\Gamma' \subseteq \Gamma$ be the subset containing the types of the first two items described above, together with the the ρ_js of the last one. The total order induced by a path $\hat{s}_i$ on s_i allows to identify each of the memory areas with the position they have in $\hat{s}_i$[4]. Let $\iota_{\hat{s}_i} : M^* \longmapsto \mathbb{N}$ be defined as follows:

$$\iota_{\hat{s}_i}(k) = \begin{cases} n, & \text{if } k \text{ is the } n\text{th element of } \hat{s}_i \\ undefined, & \text{otherwise.} \end{cases}$$

[4] Only if the program has been already proved to not throw any runtime memory-related errors.

$\iota_{\hat{s}_i}(k)$ is a labeling function for memory areas accessed in $\hat{s}_i$ that identifies every memory area with the position it has in the sequence. Depending on the path $\hat{s}_i$ taken in consideration, the domains from which the variables draw their values must be adjusted, to rule out values that do not appear in $\hat{s}_i$.

The constraints are given by $Pre(\tilde{\sigma}_b^u)$ which are unary and binary relations. Consider then the sets $J_{\hat{s}_i}$, where $K \in \wp(M^*)$ and defined as

$$J_{\hat{s}_i}(K) = \{w \in \mathbb{N} \mid \exists k \in K.\iota_{\hat{s}_i}(k) \text{ is defined} \wedge w = \iota_{\hat{s}_i}(k)\}$$

The above set is the extension of $\iota_{\hat{s}_i}$ to subsets of M^*.

4.2.2 Example. Let $\langle T, \Theta_A, \Theta_I \rangle$ as defined in Example 4.1.1 above. There are two possible paths from invocations to `u`. Consider the path $\hat{s} = \langle \texttt{Imm}, \texttt{M1}, \texttt{M2} \rangle$. The vector of variables $X = \langle this, a, b, r_1, r_2 \rangle$ is obtained from the specification and the method signature of `u` (see Fig.1, the formal parameters `A a` and `B b` and the instance on which the method is invoked) and the correspondence between memory areas and `Runnable`s (r_1 and r_2). The domains of such variables are obtained from the ordering induced by $\hat{s}$:

$$\begin{aligned} D_{r_1} &= J_s(\Theta_A(\texttt{R1})) = \{0\} \\ D_{r_2} &= J_s(\Theta_A(\texttt{R2})) = \{0\} \\ D_a &= J_s(\Theta_A(\texttt{A})) = \{1\} \\ D_b &= J_s(\Theta_A(\texttt{B})) = \{2\} \\ D_{this} &= J_s(\Theta_A(\texttt{D})) = \{2\} \end{aligned}$$

With the structure $\langle T, \Theta_A, \Theta_I \rangle$, a unit u to test and a path $\hat{s}_i$ it is then possible to translate the problem of finding possible test memory contexts in a CSP problem $\mathcal{P} = \langle X, D, \mathcal{C} \rangle$ as follows:

- Define variables $X = \langle x_1@\texttt{C}_1, x_2@\texttt{C}_2, ..., x_n@\texttt{C}_n \rangle$, each x_j with $j \in \{1...n\}$ corresponding to:
  - formal parameters in `u`'s signature
  - visible reference types mentioned in $Pre(\sigma_b^u)$
  - `Runnable`s associated with all $\mu_h \in s_i$

  each of them of class $\texttt{C}_j$.
- Define domains $D = \langle J_{\hat{s}_i}(\Theta_A(\texttt{C}_1)), J_{\hat{s}_i}(\Theta_A(\texttt{C}_2)), ..., J_{\hat{s}_i}(\Theta_A(\texttt{C}_n)) \rangle$
- Define constraints set $\mathcal{C}$:
  - for each expression in $Pre(\tilde{\sigma}_b^u)$ of the form `\inOuterScope(c,d)` add the constraint $x^c \leq x^d$, where x^c, x^d appear in X and are the variables associated with arguments or field c, d evaluate to;
  - for each expression in $Pre(\tilde{\sigma}_b^u)$ of the form `\inImmortalMemory(c)` add the constraint $x^c = 0$ (since immortal memory always has identifier 0, as it labels the root of T).

4.2.3 Algorithm. The iterative process is as follows, given $\langle T, \Theta_A, \Theta_I \rangle$ and a method u such that $\Theta_I(u) = K$, and one specification case $Pre(\sigma_b^u)$.

- For all m in K:
  - For each occurrence o of m in T
    1. let $\hat{s}_i$ the sequence describing the path from o to the root of T
    2. Translate the problem from the triple $\langle \langle T, \Theta_A, \Theta_I \rangle, u, Pre(\sigma_b^u), \hat{s}_i \rangle$as in 4.2.2
    3. Solve the problem and generate a context for all the possible solutions.

4.3 Test Case Generation

The result of the analysis will yield the information on how to build the initial state in which the tests have to be executed, thanks to the invertibility of $\iota_{\hat{s}_i}$ (where it is defined). This can be factored away in the code since several test cases will share the same code. A context given by a resulting tuple is part of the test input of the unit under test u, but it is not local (in the sense that it cannot be inferred by the specification and implementation of u) since it has been obtained by a program-wise analysis. The generated context will then have to be connected with the test cases produced by the automated test case generation tool.

4.3.1 Example. Let `u` be a method of class `D` along with its JML specification and signature:

```
Class D{
  /*@
    @ public normal_behavior
    @ requires \inOuterScope(a,b);
    @ ensures \result == 3;
    @*/
  public int u(A a,B b){...}
}
```

Continuing from Example 4.2.2, the CSP problem $\langle X, D, \mathcal{C} \rangle$ is given by

$$X = \langle this, a, b, r_1, r_2 \rangle$$
$$D = \langle D_{this}, D_a, D_b, D_{r_1}, D_{r_2} \rangle$$
$$\mathcal{C} = \{a \leq b\}$$

The corresponding resulting configuration obtained by the solution of the above problem is $\tilde{X} = \langle 0, 0, 1, 2, 2 \rangle$. A pictorial representation is given in Fig.5, along with a snippet of a possible test implementation.

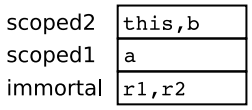


Fig. 5. Resulting test configuration for path $s = \langle \texttt{Imm}, \texttt{M1}, \texttt{M2} \rangle$ of length 3.

```
Runnable r1,r2;
LTMemory scoped1,scoped2;
@Begin
public void fixture(){
  scoped1 = new LTMemory(1000);
  scoped2 = new LTMemory(1000);
  r1 = new Runner1();
  r2 = new Runner2();
}

class Runner1 implements Runnable{
  public void run(){
    A a = new A(...);
    scoped2.enter(r2);
  }
}

class Runner2 implements Runnable{
  public void run(){
    B b = new B(...);
    D d = new D(...);
    int ret = d.u(a,b);
    Assert.assertEquals(ret,3);
  }
}

@Test
public void test(){
  scoped1.enter(r2);
}
```

5 Conclusions, Remarks and Future Work

This paper presented a method to help generate assignments that augment the test generation capabilities of a verification-based test case generator for a restricted class of Safety-Critical applications. It uses a step of static analysis and uses a translation to a CSP instance to generate all the memory stacks and allocation scenarios possible in the program for each stack, knowing the mapping from created objects to memory areas. In principle there can be programs in

which the generation would not result in a narrowing of all the possible scenarios. By the way, from the guidelines in [7,17] and the only (to the author's knowledge) RTSJ benchmarks [12], it seems hard to have a real-time Java application with such a level of complexity. The presented work does not cover what is not mentioned in the specification and in the signature of a method. Namely, if object `o` has a field `o.f` of a certain reference type, and there is no information of the position `o.f` should have in the scope stack, then nothing can be said when creating the initial state besides just assuming by default that it has to be in a more inner scope than `o`. One possible solution might be to add as much variables as needed, or otherwise to signal this as a lack of detail in the specification – with the risk of causing overspecification, or simply lots of ignored warnings.

The major drawback of this approach is that modifications to the program might lead to repeat the whole context generation process, as a test suite would be at least partially invalidated if the underlying program changes (if for instance the modifications will yield a new result for the static analysis phase).

An extension to the presented work might be to allow in $Pre(\tilde{\sigma}_b^u)$ also constraints of the type `\currentMemoryArea ==` $memoryArea\_N$, or (in the opposite direction, from code to specification) to derive, based on the the static analysis performed, a refinement to the specification in a separate `.jml` file, to reflect the information found. This would make the specification less modular and dirty because of the ways the scoped memory area objects of interest might be mentioned in the specification; but it would also be a clean way to communicate the results of the proposed procedure to the KeYVBT tool. In the author's opinion it is advisable to present together with this extension a better defined set of guidelines and patterns [18] or even a library to develop safety critical applications, in order to normalize such constraints among all possible applications of the same family. Another issue that needs to be investigated is how to connect the generated test cases with known testing criteria. An implementation of the proposed method will come shortly.

References

1. Java card, http://java.sun.com/javacard
2. Real-Time Specification for Java, http://www.rtsj.org
3. The JUnit tool, http://www.junit.org
4. The KeY-Project, http://www.key-project.org
5. The KeYVBT tool, http://www.key-project.org/download/#key-test
6. HIJA Safety-Critical Java Proposal (2006),
   Available at http://www.aicas.com/papers/scj.pdf.
7. aicas GmbH: JamaicaVM 3.4 User Documentation
8. Engel, C.: Verification Based Test Case Generation. Master's thesis, Universität Karlsruhe (aug 2006)
9. Engel, C.: Deductive Verification of Safety-Critical Java Programs . Ph.D. thesis, Fakultät für Informatik, Institut für Theoretische Informatik (ITI), Karlsruhe, Germany (2009)

10. Engel, C., Hähnle, R.: Generating Unit Tests from Formal Proofs. In: Gurevich, Y., Meyer, B. (eds.) Proceedings, 1st International Conference on Tests And Proofs (TAP), Zurich, Switzerland. LNCS, vol. 4454. Springer (2007)
11. Hu, E.Y.S., Jenn, E., Valot, N., Alonso, A.: Safety critical applications and hard real-time profile for Java: a case study in avionics. In: JTRES '06: Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems. pp. 125–134. ACM, New York, NY, USA (2006)
12. Kalibera, T., Hagelberg, J., Pizlo, F., Plsek, A., Titzer, B., Vitek, J.: Cdx: a family of real-time java benchmarks. In: JTRES '09: Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems. pp. 41–50. ACM, New York, NY, USA (2009)
13. Leavens, G.T.: JML Reference Manual, available at http://www.eecs.ucf.edu/ leavens/JML/jmlrefman/
14. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: a behavioral interface specification language for Java. SIGSOFT Softw. Eng. Notes 31(3), 1–38 (2006)
15. Leavens, G.T., Cheon, Y.: Design by Contract with JML (2004)
16. Nilsen, K.: A Type System to Assure Scope Safety Within Safety-Critical Java Modules. In: JTRES '06: Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems. pp. 97–106. ACM, New York, NY, USA (2006)
17. Nilsen, K.: Guidelines for Scalable Java Development of Real-Time Systems. Aonix (2006), available at http://research.aonix.com/jsc
18. Pizlo, F., Fox, J.M., Holmes, D., Vitek, J.: Real-Time Java Scoped Memory: Design Patterns and Semantics. Object-Oriented Real-Time Distributed Computing, IEEE International Symposium on 0, 101–110 (2004)
19. Rossi, F., van Beek, P., Walsh, T. (eds.): Handbook of Constraint Programming. Elsevier (2006)
20. RTCA: Software Considerations in Airborne Systems and Equipment Certification (1992)
21. Schoeberl, M., Sondergaard, H., Thomsen, B., Ravn, A.P.: A Profile for Safety Critical Java. Object-Oriented Real-Time Distributed Computing, IEEE International Symposium on 0, 94–101 (2007)
22. Siebert, F.: Proving the Absence of RTSJ Related Runtime Errors through Data Flow Analysis. In: JTRES '06: Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems. pp. 152–161. ACM, New York, NY, USA (2006)
23. Wellings, A.: Concurrent and Real-Time Programming in Java. John Wiley & Sons (2004)
24. Zhao, T., Noble, J., Vitek, J.: Scoped Types for Real-Time Java. In: RTSS '04: Proceedings of the 25th IEEE International Real-Time Systems Symposium. pp. 241–251. IEEE Computer Society, Washington, DC, USA (2004)

Towards Testing a Verifying Compiler*

Thorsten Bormer[1] and Markus Wagner[2]

[1] Institute for Theoretical Computer Science,
Karlsruhe Institute of Technology, Germany
bormer@kit.edu
[2] Department 1: Algorithms and Complexity,
Max Planck Institute for Informatics, Germany
mwagner@mpi-inf.mpg.de

Abstract. In this paper, we present our approach on testing a particular verification system that is industrially used to generate mathematical proofs of the correctness of C programs.
Normally, the tools used in such a verification process are seldomly verified nor thoroughly tested, and their correctness is taken for granted. Our approach to obtain assurance in such tools does not rely on the knowledge of their internal details and enables regular users of these tools to write test cases for them. Those tests are then assessed using our domain-specific *axiomatization coverage* that measures the impact of the axiomatization, which is an integral component of the verification process. Furthermore, we explore several sources of test cases, as the risk of constructing buggy test cases is high due to the input domain's complexity.

Keywords: Software validation, black-box testing, large software system

1 Introduction

Employing formal methods in the software development process is a viable, if sometimes deemed as costly, way to enhance the quality of the resulting product. One of the possibilities to use formal methods is in the verification phase of software development, supplementing the testing effort by formal software verification. Through formal verification, one obtains a mathematical proof that the program is correct with respect to its given specification.

The benefit of such a correctness proof is most apparent with safety-critical software. Additionally, in a certification process with high requirements on software quality and associated evidence of former (for example, in CC EAL 7+ or the upcoming DO178-C standard), these correctness proofs would be a valuable resource. To use the correctness proof in some certification process, the tool that was employed to generate the proof has itself to be validated in some cases.

Unfortunately, this is often not the case with existing software verification tools. As a user of these tools, internal details or even the source code of the tools are often not

* Work partially funded by the German Federal Ministry of Education and Research (BMBF) in the framework of the Verisoft XT project under grant 01 IS 07 008. The responsibility for this article lies with the authors.

available to be able to verify that the tool is working correctly—even if access to the source code is possible, lack of resources impede the application of formal methods to the tools themselves, due to the complexity of the tools.

In this paper we propose a different approach to obtain assurance in the correctness of the software verification tools, namely by testing. Our method does not rely on the knowledge of internal details of the verification tool and enables regular users of these tools to write test cases for them.

This paper is structured as follows. First, our subject under test, the verification system VCC, is presented in Section 2 with details on the toolchain and verification methodology. Then, in Section 3, the theoretical aspects of testing verifying compilers are investigated. In the subsequent sections, the theoretical results are applied to the subject under test. For this, a suitable technique for testing VCC is chosen in Section 4, where we define the domain-specific *axiomatization coverage* as our test metric, and explore several sources for test cases. Finally, the results of the testing process are presented and assessed in Section 5.

2 A Typical Verifying Compiler

For the rest of this paper, we have chosen the VCC tool [9,10], developed by Microsoft Research, as the verification system to be tested. This tool is developed in the context of the Verisoft XT project where it is successfully used within two subprojects to verify functional properties of system software.

VCC is chosen here as a particular instance of formal software verification tools that follow the "verifying compiler" paradigm. While the tool description in the following is concerned with the details of VCC, the design and architecture of VCC is similar to other tools in this area, for example Caduceus or Krakatoa, so the testing methodology of our paper is not restricted to this particular setup. VCC is being developed as an industrial-oriented verification environment for low-level concurrent system code written in C. It takes a program that is annotated with function contracts, state assertions, and type invariants, and attempts to prove the correctness of these annotations.

In the following we will give a short overview on the verification workflow and give a description of the internal architecture of the VCC tool. The particular elements of the VCC specification language and methodology are not contained in this section, but described together with the examples presented later on, as far as needed. For a thorough introduction into the VCC methodology, see [9].

2.1 The VCC Workflow

To verify whether a program fulfills certain functional properties using VCC, the intended properties are first formulated with the help of the VCC specification language, such as method contracts or invariants on data types. This specification language is similar to those found in ESC/Java2 [11], Spec#, and HAVOC [8]. As in all these systems, the program's specification is stored as inline source code annotations. These annotations are invisible to a normal C compiler (making use of the C preprocessor features) but are analyzed by VCC within the verification process.

Invoking VCC on an annotated C source file has one of the following outcomes: (a) VCC reports that the program fulfills its specification as given by the annotations or (b) VCC could not prove that the program meets the specification. The latter case may have several reasons (for example, not enough system resources for the prover, a bug in the software or specification)—for each of the error sources, there are appropriate tools in the VCC package to inspect and debug these errors.

2.2 Architecture of the VCC-Toolchain

To prove that a program meets its specification, the VCC tool internally makes use of a toolchain of three tools: from the annotated C code, with the help of VCC's compiler, a representation in an intermediate, imperative programming language with embedded specification constructs (called *BoogiePL* [17]) is generated. This BoogiePL representation is then further processed by the Boogie tool into proof obligations. These are then proven or refuted by the Z3 theorem prover—leading to either the statement that the original program meets its specification, or, if the proof obligations are refuted, to a counterexample.

In the following, we will give a short overview on each of these steps and components in the toolchain.

VCC's compiler The VCC compiler is build by using the Common Compiler Infrastructure (CCI)[3]. Annotated C programs are read and turned into CCI's internal representation to perform typical tasks of a regular C compiler, such as name resolution, and type and error checks. Next, the fully resolved input program is subject to several transformations: (1) simplifying the source, (2) adding proof obligations that result from the methodology, and (3) finally generating Boogie code.

Boogie When a C program is analyzed and found to be valid, it is translated into a Boogie program that encodes the input program according to the employed formalization of C. Boogie is an intermediate verification language and a verification system that acts as a layer on which program verifiers for other languages can be built upon. It is used by a number of software verification tools including Spec# and Havoc.

Before the Boogie program is fed to the Boogie program verifier, which translates it into a sequence of verification conditions, the *prelude* is added, which is an axiomatization of the C intrinsic memory model, object ownership, type state and arithmetic operations. Then, the verification conditions are passed to an automated theorem prover to be proven or refuted.

Z3 Z3 [12] is a first-order theorem prover that checks whether a set of formulae is satisfiable in the built-in theories. Those cover, for example, the equality over free function and predicate symbols, real and integer arithmetic, and bit-vectors.

3 Validation of Verification Environments

3.1 Software Validation

To check whether a software system meets its specification and fulfills its intended purpose, a plethora of techniques (for example, deductive verification, static analysis,

[3] Microsoft Research: CCI. 3 May 2010 `http://ccimetadata.codeplex.com/`

and white-/black-box testing) can be applied. In this work, we have chosen to use black-box testing as a cost-effective procedure to establish assurance that our target, VCC, works correctly.

In general, functional conformance testing is classified as a black-box approach when an external tester can only observe the outputs generated by the implementation upon the receipt of inputs, without any information about the internal design of an implementation. Conformance is the relation between a specification and an implementation, and the relation is valid if the implementation does not present behaviors that are not allowed by the specification. If the implementation is given as a black box, only its observable behavior would be able to be tested against the required behaviors by the specification.

Towards black-box testing of verification systems, we considered the following approaches to be applicable. *Error guessing* is an ad-hoc approach mostly based on experience. *Equivalence Partitioning* can be applied when the domain of each input parameter of a function is structured into equivalence classes. *Boundary Value Analysis* assumes that errors tend to occur near extreme values because typical programming errors—for example, wrong termination conditions for loops—are often related to these boundaries. *Model-driven testing* [2] was not considered applicable because of the very costly process of constructing a model for large systems. This is the case for verification systems, where the input and output data is tightly coupled to the behavior specifications of the verification system.

3.2 Validation Techniques for Verification Systems

When it comes to identifying the components of the verification system that are to be validated, we identified two major obstacles. The first one was the *complexity of the toolchain*. Verification systems are usually large software systems: they are composed of complex parsers for the input languages, mechanisms to rewrite the input into proof obligations, and possibly problem solvers and other tools. The second obstacle was the *complexity of the supported languages*. Automatic verification systems usually support a programming language that is annotated with elements from a specification language. This results in the complex interaction of elements from both languages.

The complexity of the toolchain can be countered by testing the components individually, if possible. A structured divide-and-conquer approach towards the interaction of language elements cannot be defined as straightforward. This is due to the rather unstructured input domain of a verification system; each test is not simply a combination of some values for a function to be tested, but an entire C program including annotations. Some structures within the domain can be achieved by defining some orders over the individual language elements, or by aggregating elements, such as “arithmetic operators” and “memory model specific operators”, to domains. Based on these domains, test cases can be created systematically by using the combinatorial testing approach. Once a thorough test is performed, combinatorial testing offers an easy and intuitive evaluation of the testing process itself: based on the structured approach, the coverage on n-wise coverage combinations can be computed, and these numbers can help to build trust in the tool.

Related Work In principle, instead of using our testing approach, parts of the verification tools available could be formally verified by using either the verification tools themselves or others. There have been several efforts to develop completely certified program verifiers, e.g., in the Bali project [21], the LOOP project [15], and in the Mobius project [4]. Several times, tricky verification examples were proposed to test verification tools ([14]), and furthermore, components of Java verification tools were verified ([1]). One example of such a soundness proof conducted is the verification of the rewrite rules of Caveat's[4] integrated theorem prover by using PVS[5]. In addition, though, also all combinations of C's syntactic constructs were tested. Due to our limited resources, a comparable approach could not be realized in our scenario.

In their discussion on whether verification systems and calculi have to be verified in general, Beckert and Klebanov [6] argued that in practice, a more powerful and sufficiently correct system may be used in favor of a less powerful but correct system. Although they considered the verification of the tools or its components as important, they advised the developers of verification systems to test more frequently.

A less labor-intensive method than (cross-)verifying parts of the verification systems would be conducting (cross-)validation of the components by comparative testing. However, the question is whether such a comparable (verification) system exists. For the part of programming language, regular compilers may be used as sources for comparative statements on the parsability of source code, but finding several verification systems with similar features that are able to produce comparable outputs from the same source code is a problem.

Regarding the annotation languages that are commonly used, we observed the relative similarity between languages such as Java Modeling Language [7], the ANSI/ISO C Specification language (ACSL)[6], and Microsoft's variants. With possible convergence of specification languages in the future, we expect the number of comparable verifications systems to increase. Thus the creation of verification-specific test cases will become more desirable because of their increased reusability.

Concentrating on the theorem provers that are used in the last stage of VCC to discharge verification conditions, different approaches are capable of building trust in them. For example, cross-validation can be used, based on established problem libraries such as the well-known TPTP library[7]. Alternatively, the results can be validated by using proof checkers. One example of such a system is the Formally Verified Proof Checker that was implemented in ML and even formally verified by using HOL88 [22].

An interesting application of conformance testing is the official validation test suite for FIPS C (a dialect of C).[8] In order to determine the coverage on the language standard of a test suite, the language standard itself was implemented in a so-called *model*

[4] CEA-LIST: The Caveat Tool. 3 May 2010 `http://www-list.cea.fr/labos/gb/LSL/caveat/index.html`

[5] SRI International: PVS. 3 May 2010 `http://www.csl.sri.com/projects/pvs/`

[6] CEA-LIST/INRIA-Sacley: ACSL. 3 May 2010 `http://frama-c.com/acsl.html`

[7] Geoff Sutcliffe, Christian Suttner: The TPTP Problem Library for Automated Theorem Proving. 3 May 2010 `http://www.cs.miami.edu/~tptp/`

[8] Derek Jones: Who Guards the Guardians? 3 May 2010 `http://www.knosof.co.uk/whoguard.html`

implementation, i.e., an actual compiler based on the language description. Statements of the model implementation were then mapped back to the standard, allowing the authors to show that all of the requirements in the standard were implemented. With this approach, statement coverage w.r.t. this model implementation thus relates to coverage of the language standard elements. In the end, a statement coverage of 84% of the model implementation was achieved by a comprehensive test suite, demonstrating that the test suite checks a substantial portion of the C programming language.

4 Testing of VCC

To effectively apply software testing to VCC, we have to first identify the important quality attributes of the subject under test. These attributes can then be used to derive or select useful metrics in order to assess the quality of testing. Then, we discuss several possible sources for test cases in order to apply the concept of cross-validation to verification environment testing. It has to be noted that our approach is only weakly related to "regular" compiler testing, as our focus is not on the parsing capabilities and automatic error corrections, but on VCC's design goal, i.e., the ability to fully automatically prove a program's correctness.

4.1 Test Objective

In the following we concentrate on the soundness of verification systems. The discussion of Beckert et al. [5] on the completeness of verifying compilers is related to this definition, in which they distinguished between different types of annotations. For example, it can be the case that a program is correct with respect to its requirement specification, but the toolchain is unable to prove it. The reason for this is the missing *auxiliary* annotations that would have guided the theorem prover to the correctness proof. In the following, the term *annotations* is used to cover all the requirement specification annotations and the auxiliary annotations of a program.

From VCC's point of view, there is no way of differentiating a test case that is supposed to succeed from one that is supposed to fail. It is important to remember that we are not in the situation of verifying annotated programs, but of observing VCC's verification attempts on annotated programs. This leaves us with the following classification of test cases: (1) *successful* cases, where the outcome of a verification attempt equals the expected outcome, and (2) *failing* cases, where the outcome of a verification attempt does not equal the expected outcome. Hence, a test case for a verification system consists of an annotated program and some expected output. The verification system itself is not part of the test case; components such as the axiomatization remain unchanged for the testing process.

4.2 Testing with Respect to the Axiomatization

As already mentioned in Section 2.2, VCC's prelude contains an axiomatization of C written in Boogie. And within the multi-stage verification process, it has a significant impact on the outcome. Furthermore, the prelude is accessible to a human analyst, both on the code level and on the level of understanding the effects of the prelude's elements.

Based on the importance and its accessibility, we chose to test VCC with respect to the prelude, that is, to observe the impact of the prelude on the verification process. Alternatives will be discussed in Section 4.3.

The prelude itself is a Boogie program. In general, a Boogie *program* consists of a *theory* that is used to encode the semantics of the source language, and an *imperative part* [3]. A theory is composed of type declarations (*keyword: type*), symbol declarations (*const*, *function*), and axioms (*axiom*). The imperative part of a Boogie program consists of global variable declarations (*var*), procedure headers (*procedures*), and procedure implementations (*implementations*). The size of the prelude in our case is about 2900 lines of code—for easier maintenance and improved legibility the prelude is further structured into sections concerning different language- and specification features. Later on, we will modify the structure of the prelude.

4.3 Coverage Measurement

In the following, we describe our approach chosen to determine the impact of the axiomatization used. The idea is to determine the subset of elements of the original prelude that is used by the test case. It is checked whether or not an element of the prelude is needed by comparing the original output of VCC with the result when the selected element is left away. If the element can be left away, it is discarded for the given test case and the process is iterated until no more elements can be left away. Note that, in general, the minimal set of prelude elements for a given program is not uniquely defined. Depending on the generated proof obligations, different sets of prelude elements may be needed[9] and thus the selection strategy when reducing the prelude matters.

Based on our approach, the straightforward definition of axiomatization coverage follows:

Definition 1. *Given a verification environment v, its specification s , the complete axiomatization consisting of m elements, a test case t, and a corresponding minimized axiomatization a_{vst} with n_{vst} elements. Then the* axiomatization coverage *is defined as $Cov(v, s, t) = n_{vst}/m$. For a set of test cases $T = t_1, \ldots, t_n$ and corresponding minimized axiomatizations $a_1, \ldots, a_n$, the coverage is defined as $Cov(v, s, T) = n_a/m$ where $\bigcup_n a_i$ has n_a distinct elements.*

Tests that need rather many elements of the axiomatization can be regarded as *strong tests*; on the other hand, tests that requires only a rather small number of elements can also be regarded as strong because the verification task itself may be very complex. However, the latter kind of test strength addresses the problem of "the difficulty of verification", rather than "the interaction of the prelude's elements", in which we are actually interested.

Discussion and Alternatives In his work, Littlefair [19] examined the relation between the consideration of quality in software engineering and software metrics. Based

[9] e.g., consider the proof obligation $a \vee b$: either all elements needed to prove a are needed or all relevant elements for b

on his observations and on Weyuker's proposed conditions for useful measures of software complexity [24], the usefulness of our own measurement can be evaluated—exemplarily, we discuss two conditions.[10] One of the conditions requires that "there exist programs P and Q such that $|P| \neq |Q|$". This requirement motivates that for a measure to have any value at all, it has to enable some discrimination between different programs. Axiomatization coverage fulfills this requirement. Another condition requires that "For all programs P and Q, and the program $P;Q$, which is obtained by combining P and Q, $|P| + |Q| \leq |P;Q|$". The justification for this property is the notion that the interaction between parts of a program may introduce complexity, additional to that present in the components. Hence, the amount of added complexity may only be non-negative. Axiomatization coverage does not fulfill this requirement: due to significant overlap in the minimized preludes needed by two programs, $|P| + |Q| \leq |P;Q|$ usually does not hold. Despite not fulfilling several of Weyuker's conditions[11], our definition of axiomatization coverage has the advantage that test cases can easily be compared because the result of the coverage computation is a single number. While allowing for a quick classification of test cases into *comprehensive* test cases and test cases with *limited scope*, it does not take into consideration *which* elements of the prelude are needed.

Alternative 1. As the first alternative, we suggest to count the number of the covered equivalence classes of "language feature", for example, (1) *array indexed with negative value* vs. *array indexed with the value 0*, (2) *unwrapping an object that is not wrapped* vs. *unwrapping a wrapped object*. In spite of its obvious benefits attributed to the strong relationship to the boundary value analysis, we do not expect it to be measured easily if automatically at all, as it is not clear what a language feature is.

Alternative 2. Going away from the axiomatization coverage and back to the idea of covering language features, a metric can be used that counts the language features used by a test case. In fact, this can be refined by counting the features of the programming language and the features of the annotation language separately. Based on combinatorial testing, an extension of this metric would be the use of the number of "t-wise language feature element combinations" that are covered by a test case. After testing, a high value of this extended metric would allow for a high trustworthiness in the subject under test, as data reported in several studies ([23,20,16]) show that software failures in a variety of domains were caused by the combinations of several conditions.

Alternative 3. Similarly to the last alternative, and by adapting the derived metric LOC/COM (lines of code per line of comments, interpretable as *maintainability*), it is possible to use LOA/LOC, where *LOA* is the number of lines of annotations needed. Thus, it is possible to estimate how many annotations are needed to verify a given code block. A high ratio may indicate code that is difficult to verify, while a low ratio may indicate easily verifiable code.

Further alternatives are imaginable, but as the way of defining the metric gets more complicated, it becomes less conclusive how to actually interpret the results. In general,

[10] In Weyuker's notation the letters P, Q, R represented distinct programs, and the result of the adequacy measurement was signified by $|P|$, $|Q|$, $|R|$.

[11] The discussion was omitted because these are all conditions based on the composition of programs, which cannot be fulfilled due to the overlap in necessary prelude elements.

it is very likely that a single metric is never able to be presentable as a single expression for software quality because the objectives targeted by the models of software quality tend to be multi-dimensional and hierarchical.

4.4 Sources of Test Cases

In the following, we address the issue of producing meaningful test cases, as the systematic creation of a large number of meaningful tests is not trivial. To reduce the impact of erroneous test cases on the testing process, we obtain the test cases from three independent sources, as a form of cross-validation.

At first, we explored the possibility of using the official C language standard. In the next step, we analyzed existing C compiler test suites and investigated ways of adapting those tests. Finally, tests from other verification systems were analyzed for their possible adaption.

C Standard The C standard ISO/IEC 9899:201x, often called *C1X*, was selected to be the first source of possible tests. Although it does not include tests, it specifies the form and establishes the interpretation of C programs. The standard uses the Backus-Naur form for the syntax and prose for the semantics and constraints.

Due to the implementation details of the VCC toolchain, two variations of the C language had to be considered that deviate partially in language features supported compared with C1X. For neither a comprehensive lists of the supported C language features are available, leading to only partially usable C test cases.

In the following, we demonstrate how test cases can be constructed, based on the sources of information on the supported C standards, based on the first paragraph of C1X's section 6.3.2.3 on pointers:

> **6.3.2.3 Pointers**
> A pointer to `void` may be converted to or from a pointer to any incomplete or object type. A pointer to any incomplete or object type may be converted to a pointer to `void` and back again; the result shall compare equal to the original pointer.[...]

Based on the information gathered from the standard we created a single test file for this paragraph. Exemplarily, we annotated the test file with as much information as possible; that is, we have listed the motivational source for the tests, the links to supplementary information, and we have quoted the sentences from the standard's paragraph. Thus, we achieve a strong link between the standard and the derived tests. An excerpt of the full version, which was successfully verified by VCC in accordance with the standard, is presented below:

```
1  //Scope: C1X 6.3.2.3 Pointers, p. 61f
2  #include "vcc.h"
3
4  void function6323_1(void) {
5    //object types:
6    char* b;  [...]
7
8    //paragraph 1: 1. A pointer to void may be converted to or from a pointer to
9    //any incomplete or object type.
10   b = (char*)v; v = (void*)b;  [...]
11
12   //paragraph 1: 2. A pointer to any incomplete or object type may be converted
```

```
13  //to a pointer to void and back again; the result shall compare
14  //equal to the original pointer.
15  assert(b == (char*)(void*)b);  [...] }
```

C Compiler Test Suites With VCC being a special kind of compiler, the construction of test cases using compiler test suites is a straightforward approach. Test suites that check conformance to standards are often called validation suites, and those validation suites are very influential. Several commercial validation suites are available on the market (for example the ACE SuperTest or the Perennial ANSI C Validation Suite), although for licensing reasons we chose to use the test suite of the GNU Compiler Collection (GCC) version 4.4.0[12]. The C compiler of GCC supports C90, and parts of C99. The C specific part (about 12,000 files) of the GCC test suite contains generic tests that are supposed to run on any target, and platform specific tests. Information on the purposes of the tests is limited, thus complicating the analysis of the test cases.

In addition to verifying developer-defined functional properties, VCC implicitly checks for undefined behavior, such as, null pointer dereferences, division by zero, over- and underflow. It does so by automatically inserting additional assertions into the verification conditions, which precede the translated operation. Because of these checks, non-annotated source code normally cannot be verified, despite the lack of explicitly stated functional properties. Therefore, minimal annotations have to be included, for example, the definition of `writes` and `reads` clauses.

We used an iterative approach to adapt files from the GCC test suite. First, we checked whether Microsoft's own C compiler can compile the source code without warnings or errors. Then, minimal annotations were added, so that VCC verifies the source code without warnings or errors. Finally, additional specification was added based on comments and a close inspection of the C code, defining pre- and postconditions, as well as invariants to capture the functional properties of the program. This step-wise approach is demonstrated in Figure 1. There, we used the information given in the `main` function to construct the postcondition. In this function, another function `f` is called first, and subsequently, the result of `f` is checked against what seems to be the expected result of `f`, that is `if (b != 9)`. If the expected result is not met, the program stops abnormally, otherwise it stops normally. Both situations return different exit codes, which are then interpreted by the test framework.

Verification Environments The motivation behind this approach is that the time intensive task of creating interesting test cases for verification environments can be saved by adapting existing test cases.

VCC. VCC version 2.1.20731.0 is deployed with a set of 400 test cases, addressing specific domains, such as *arrays*, *claims*, or *ghost code*. Again, information on the tests' purposes is very limited; however, some information on the background can be obtained by an experienced user of VCC by reviewing the source code in combination with the expected result. Out of the 400 test cases, 202 can be regarded as true positives, and 198 as true negatives. This surprisingly balanced ratio indicates that the developers of VCC use a systematic testing approach to test succeeding and failing verification.

[12] GNU Compiler Collection: 4.4. 3 May 2010 `http://gcc.gnu.org/gcc-4.4/`

```
1  #include "vcc2.h"
2  int b;
3
4  void f ()
5  writes(&b)                          //A
6  ensures(old(b)==0 ==> b==9)         //B
7  ensures(old(b)!=0 ==> b==old(b))    //B
8  {
9    int i = 0;
10   if (b == 0)
11     do
12     invariant(0<=i && i<10) {       //B
13       b = i;
14       i++;
15     } while (i < 10) ;
16 }
```

```
17 int main ()
18 writes(&b)                          //A
19 ensures(old(b)==0                   //B
20           ==> result == 0)
21 ensures(old(b)!=0 && old(b)!=9      //B
22           ==> result == 1)
23 {  f ();
24   if (b != 9) return 1;
25   return 0;
26 }
```

Fig. 1. Demonstrating the iterative adaption of the GCC test case files. The test case file `990604-1.c` without any annotations is amended with minimal annotations (lines marked with `A`), and with functional specifications (lines marked with `B`).

Spec#. The collection of verified algorithms from Leino and Monahan [18] contains 38 relatively complex real-life algorithms,[13] such as an insertion sorting algorithm and a minimal distance algorithm. The algorithms are written in C# and verified by Spec#. Furthermore, the algorithms are relatively well documented.

Frama-C/Jessie. The Framework for Modular Analysis of C programs (Frama-C)[14] is a set of program analyzers with Jessie as the deductive verification plug-in. Using the Why back-end [13], automatic theorem provers can be used to perform fully automatic verification. The C files are annotated by using ACSL (see Page 3), which is comparable to VCC's annotation language. Similar techniques are used to express, for example, method contracts, invariants of loops and data structures, and ghost code. As of the Frama-C release Beryllium 20090601, the distribution comes a set of 236 test case files for the Jessie plug-in. Compared to the Spec# tests, the translation is slightly more complex because the annotation language shares less common concepts with VCC's than Spec#'s. Still, the annotations can be very helpful, especially when invariants are provided.

Comparison of the Sources During the exploration of the different sources for test cases, we have made several observations. Based on these, the above-mentioned approaches can be compared both qualitatively and quantitatively.

The highest assurance level is given when the programming language standard is used to derive test cases. However, this is the most labor-intensive approach. Furthermore, it may be difficult to determine if a failed test is caused by a misinterpretation of the standard, or by an incorrect implementation inside the verification tool. Nevertheless, this approach may be useful when corner cases are needed.

An almost arbitrary number of test cases can be created by adopting C compiler test suites, which is less labor-intensive than the standard-based one. However, the task

[13] Rosemary Monahan: Verified Textbook Examples. 3 May 2010 `http://www.rosemarymonahan.com/specsharp/`

[14] CEA-LIST/INRIA-Sacley: Frama-C. 3 May 2010 `http://frama-c.cea.fr`

remains very time consuming, and debugging may be difficult because the C compiler and the verification tool may have different implementations of the C features.

The use of other verification tools as sources has the potential to offer substantial support to find the annotations needed for full functional verification. But the number of the transferable tests is relatively small, and it cannot be guaranteed that two different verification tools are capable of verifying the same functional properties of a program.

4.5 Test Framework

We implemented a framework that allows for the automated execution and evaluation of tests. This enables us to find errors in VCC, and to perform the regression testing of VCC and of a code base. The framework has the benefit that it can be reused without any changes at all if VCC supports further programming languages in the future. Furthermore, it can be adapted with little effort to other fully automatic verification environments, if the axiomatization used by these environments is externally modifiable. Support for interactive verification tools is not implemented, although it should be possible to some extend using capture/replay tools.

5 Test Results

In the previous section, we have laid the basis for testing the Verifying C Compiler. The theoretical background was investigated, a suitable test objective defined, and sources for test cases were explored. In this section we present the results of the actual runs of the test framework.

5.1 Prelude Coverage Results

Exemplarily, our test harness computed the axiomatization coverage that is achieved by VCC's own test suite. The used test suite contained 400 test cases, which were automatically extracted from the test suite collection files of VCC 2.1.20731.0. The harness determined the axiomatization coverage that is achieved on the axiomatization of VCC version 2.1.20731.0, and for comparative reasons the coverage on the axiomatization of VCC version 2.1.20908.0 (a version about 6 weeks further into development). The earlier axiomatization contains a total of 858 elements, of which 575 where covered. Out of these 858 elements, 186 of the 378 axioms were covered; the other elements are, for example, type and constant definitions, and helper functions. To give the reader an idea of how this axiomatization is organized: 1) the class of C language features contains 432 elements, containing 212 axioms, of which 84 were covered, and 2) the class of specification language features contains 426 elements, where 102 of the 166 axioms were covered. Regarding the latter axiomatization, it contains 896 elements in total, of which 509 were covered, and only 139 out of 384 axioms were covered.

Before the runs, we had expected that the number of covered elements would stay roughly the same because features that were added to VCC (indicated by the 38 additional elements) could not be tested by the old test cases. However, the number declined significantly from 575 to 509 elements. Investigations reveal that the reason for this is that axioms and procedures were modified, for example, by removing requirements or

by adding predicates. These changes lead to the necessary inclusion of less elements, which is observed in smaller minimized preludes. Based on the detailed information from the test runs, it is possible to establish links between the different minimized preludes required by a test case and the changes made to VCC's source code and prelude. The old tests can still be regarded as relevant to testing VCC, however, they proved to be less adequate to the newer version of VCC. This can be observed in the decline in the overall coverage from 67.0% to 56.8%.

In subsequent tests, we were sometimes able to construct tests so that the use of a specific axiom was triggered. When creating such tests, one has to deal with the complex and not visible interaction between the elements and the dependencies among themselves. Although, this approach could be used to systematically add tests to the test suite, it becomes increasingly difficult to cover previously uncovered elements.

5.2 Issues Encountered

During our investigations, we encountered several issues. For example, we discovered one bug in the axiomatization with regard to the ownership model. Furthermore, we discovered one case of VCC being more lenient than Microsoft's C compiler, as it did not care about a missing semicolon after the declaration of a C structure. Among the minor issues are the following: we discovered problems with the interaction of VCC's command-line parameters, and one problem with VCC's model viewer showing outdated values. Regarding Boogie, we encountered an incompatibility with file names starting with numbers, as they yielded Boogie identifiers with illegal characters, which in return caused Boogie to report errors.

6 Conclusions

The way verification environments are currently tested is not very satisfying. Most of the time, the tools are written by researchers, and as long as those environments are not actually used for the verification of critical systems, there is no real demand for trust in these systems. The Verifying C Compiler (VCC) is one of these tools that are being used for the verification of industrial products. In this paper, we investigated systematic approaches for the validation of verification systems. Once we had specified what it means for a verification system to be correct, we were confronted with the generation and assessment of test cases for VCC.

The input domain of such a system is the result of the manifold combination of elements from the C programming language and from the language of verification-specific annotations. We reduced the risk of constructing incorrect test cases by choosing trustworthy sources, such as, the official C language standard ISO/IEC 9899:201x, the test suite of the GNU Compiler Collection, and the test suites of the verification tools Spec# and Frama-C/Jessie. While the standard offers the highest level of trust, the derivation of test cases is the most labor-intensive one. Tests adapted from other verification tools have the potential to offer substantial support to find the annotations needed for the functional verification. But the number of those tests is relatively small, compared to those available from the compiler test suite. The latter, however, does not contain any annotations, which are not easy to come by.

When we investigated how the individual components of verification systems can be tested and how the test cases can be assessed, we realized that the common approach of measuring the percentage of the system's executed code statements would not take into account the special features of verification systems. Therefore, we defined *axiomatization coverage* as our domain-specific metric for VCC, in order to assess the test cases and to observe the impact that the individual elements of the axiomatization have on the verification process. Concerning the tests we performed, we noticed that VCC's own test suite requires only about 60% of the axiomatization. Based on this fact and on further observations, we draw the following two conclusions: (1) The used part of the axiomatization seems to correctly reflect the developers' assumptions about how the verification methodology is supposed to work, and (2) additional test cases should be written to achieve a higher coverage. If it is not possible to trigger the use of the prelude's element, the importance of this element should be reconsidered.

In the future, we plan to investigate ways of automatically annotating original C compiler test case files with as many annotations as possible. One way would be to statically analyze its abstract syntax tree (AST) and then modify it. An abstract syntax tree is a tree representation of the syntactic structure of the source code, where each node of the tree stands for a construct occurring in the source code. Once the AST is created, it can be modified, and then the tree can be unparsed to obtain the refactored source of the C program. Additionally, we plan to extend the experiments by determining the "regular code coverage" that is achieved by our set of tests. This will enable us to compare our coverage approach with those established in the software testing community.

Acknowledgments We thank the anonymous reviewers for their comprehensive and helpful comments.

References

1. W. Ahrendt, A. Roth, and R. Sasse. Automatic validation of transformation rules for Java verification against a rewriting semantics. In *12th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, volume 3835, pages 412–426. Springer, 2005.
2. P. Baker, Z. R. Dai, J. Grabowski, I. Schieferdecker, Øystein Haugen, and C. Williams. *Model-Driven Testing: Using the UML Testing Profile*. Springer, Secaucus, NJ, USA, 2007.
3. M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *3rd International Symposia on Formal Methods for Components and Objects (FMCO)*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer, 2005.
4. G. Barthe, L. Beringer, P. Crégut, B. Grégoire, M. Hofmann, P. Müller, E. Poll, G. Puebla, I. Stark, and E. Vétillard. MOBIUS: Mobility, ubiquity, security. volume 4661 of *Lecture Notes in Computer Science*, pages 10–29. Springer, 2006.
5. B. Beckert, T. Bormer, and V. Klebanov. On essential program annotations and completeness of verifying compilers, 2009. unpublished.
6. B. Beckert and V. Klebanov. Must program verification systems and calculi be verified? In *3rd International Verification Workshop (VERIFY), Workshop at Federated Logic Conferences (FLoC)*, pages 34–41, 2006.

7. L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. 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*, 7(3):212–232, June 2005.
8. S. Chatterjee, S. K. Lahiri, S. Qadeer, and Z. Rakamaric. A reachability predicate for analyzing low-level software. In *13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 4424 of *Lecture Notes in Computer Science*, pages 19–33. Springer, 2007.
9. E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 5674 of *Lecture Notes in Computer Science*, pages 23–42. Springer, 2009.
10. E. Cohen, M. Moskal, W. Schulte, and S. Tobies. Local verification of global invariants in concurrent programs. In *Computer Aided Verification (CAV)*, Lecture Notes in Computer Science. Springer, 2010. To appear.
11. D. R. Cok and J. R. Kiniry. ESC/Java2: Uniting ESC/Java and JML. *Lecture Notes in Computer Science*, 3362:108–128, 2005.
12. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis (TACAS)*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.
13. J.-C. Filliâtre and C. Marché. The why/krakatoa/caduceus platform for deductive program verification. In *Computer Aided Verification, 19th International Conference (CAV)*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177. Springer, 2007.
14. B. Jacobs, J. Kiniry, and M. Warnier. Java program verification challenges. *Lecture Notes in Computer Science*, 2852:202–219, 2003.
15. B. Jacobs and E. Poll. Java program verification at Nijmegen: Developments and perspective. *Lecture Notes in Computer Science*, 3233:134–153, 2004.
16. D. R. Kuhn, D. R. Wallace, and A. M. Gallo. Software fault interactions and implications for software testing. *IEEE Transactions on Software Engineering*, 30(6):418–421, 2004.
17. K. R. M. Leino. This is Boogie 2, 2008. Working draft 24 June 2008.
18. K. R. M. Leino and R. Monahan. Automatic verification of textbook programs that use comprehensions. In *Workshop on Formal Techniques for Java-like Programs (FTfJP)*, 2007.
19. T. Littlefair. *An Investigation into the Use of Software Code Metrics in the Industrial Software Development Environment*. PhD thesis, Edith Cowan University Mount Lawley, 2001.
20. V. Nair, D. James, W. Ehrlich, and J. Zevallos. A statistical assessment of some software testing strategies and application of experimental design techniques. *Statistica Sinica*, 8(1):165–184, 1998.
21. D. von Oheimb. Hoare logic for Java in Isabelle/HOL. *Concurrency and Computation Practice and Experience*, 13(13):1173–1214, 2001.
22. J. von Wright. The formal verification of a proof checker. SRI internal report, 1994.
23. D. R. Wallace and D. R. Kuhn. Failure modes in medical device software: An analysis of 15 years of recall data. *International Journal of Reliability, Quality, and Safety Engineering*, 8(4), 2001.
24. E. J. Weyuker. Evaluating software complexity measures. *IEEE Transactions on Software Engineering*, 14(9):1357–1365, 1988.

Dynamic Frames in Java Dynamic Logic

Peter H. Schmitt, Mattias Ulbrich, and Benjamin Weiß

Karlsruhe Institute of Technology
Institute for Theoretical Computer Science
D-76128 Karlsruhe, Germany
{pschmitt,mulbrich,bweiss}@ira.uka.de

Abstract. In this paper we present a realisation of the concept of dynamic frames in a dynamic logic for verifying Java programs. This is achieved by treating sets of heap locations as first class citizens in the logic. Syntax and formal semantics of the logic are presented, along with sound proof rules for modularly reasoning about method calls and heap dependent symbols using specification contracts.

1 Introduction

To successfully support modular verification of object-oriented software, it is essential to be able to define relevant portions of memory and reason about the effects of method execution on them. Portions of memory, i.e., sets of heap locations, are called *frames* in this context or—since they themselves are subject to change during program execution—*dynamic frames*. The theoretical concept of dynamic frames was introduced in [7] and first implemented in [21] and later in [10]. Specification with dynamic frames is related to the use of data groups [11], separation logic [16, 20], and to approaches based on ownership types [1, 15].

In this paper we investigate the integration of the dynamic frames specification style into the verification of sequential Java programs based on *dynamic logic* [5]. In many verification methods, the task of verifying that a property φ holds after execution of a program `p` is solved by successively computing *weakest preconditions* [4] in first-order predicate logic of parts of the program starting from its end. In dynamic logic, the weakest precondition can be directly written, thanks to the modal operator $[\cdot]$, as the formula $[\mathtt{p}]\varphi$. Dynamic logic can be augmented with a symbolic representation of state changes called *updates* [18]. This extension allows giving inference rules for dynamic logic that compute (first-order) weakest preconditions by performing a *forward symbolic execution* of the program `p` starting from the beginning. The proof tree that unfolds by successive applications of these rules will eventually contain only first-order proof subgoals. This form of verification is the foundation of the KeY system [2]. Dynamic logic is also used for Java verification in the KIV system [22].

An issue in program verification to be addressed no matter how proof obligations at the program level are transformed to first-order proof goals is the representation of the heap. In a closed-world setting, where the entire program is known at verification time, an explicit heap representation can be dispensed

with, saving some complexity. This was e.g. realised in the KeY system. In a modular setting, where one strives for abstract specification of interfaces and local reasoning, the situation is different: here, reasoning about which frame is changed by a program, or about which frame the execution of a program depends on, becomes crucial. In this setting, the flexibility provided by an explicit representation of the heap seems to offer decisive advantages.

In Sect. 2 we motivate the use of dynamic frames with a simple example. The dynamic logic to be presented will explicitly represent dynamic frames as sets of locations. Syntax and semantics and some exemplary proof rules of this logic are given in Sect. 3. Contract-based proof obligations and proof rules for verifying dynamic frames specifications are defined in Sect. 4. Conclusions in Sect. 5 wrap up the paper.

2 Motivating Example

As an example, we consider the Java program shown in Fig. 1. The intention behind the `List` interface is that objects of this type represent lists of objects. The interface provides methods for querying the size of the list, retrieving an element out of the list at a given index, and appending an element to the end of the list. Class `ArrayList` implements the interface with the help of an array, and class `Client` is an artificial snippet of client code using the interface.

Our goal is to specify this program following the *design by contract* paradigm [14]. That is, we are interested in providing *pre- and postconditions* for the methods of the program, where we refer to a pair of a pre- and a postcondition as a *method contract*. Furthermore, the goal is to *verify* the correctness of these contracts using dynamic logic, and to do so in a *modular* (or *local*) fashion: the verification of a given method should not make use of implementational details that are not visible in this method. For example, when verifying `m` in `Client`,

Java

```
interface List {
  int size();
  Object get(int i);
  void add(Object o);
}
class Client {
  public int x;
  Object m(List l) {
    x++;
    return l.get(0);
  }
}
```

```
class ArrayList implements List {
  private int n = 0;
  private Object[] a = new Object[10];
  public int size() {
    return n;
  }
  public Object get(int i) {
    if(0 <= i && i < n) return a[i];
    else return null;
  }
  //method "add" omitted
}
```

Java

Fig. 1. Example program

we do not want to make use of the fact that there is only one implementation of `List`, nor of the internals of this particular implementation. Instead, reasoning about the dynamically bound call to `get` should be based only on the contract for `get` in the interface. For subtypes of the interface, we only require that all overriding method bodies satisfy the contracts given at the level of the interface; this means that we enforce *behavioural subtyping* [12].

A main difficulty in specifying an interface such as `List` is that we do not have access to any implementational data structures for writing our specifications. The general solution is to use *data abstraction* [6]: we specify the interface in a more abstract fashion, using either some form of *abstract fields* (sometimes called *model fields* [3]), or side-effect free methods present in the program. Here, we choose to specify `get` with the help of the `size` method, and with the help of an abstract Boolean field *inv*:

$$\textit{pre}\colon \texttt{this}.\textit{inv} \land 0 \leq \texttt{i} \land \texttt{i} < \texttt{this.size()} \qquad \textit{post}\colon \texttt{res} \doteq\!\!\!\neq \texttt{null}$$

We use a dot to distinguish some syntactic operators of the logic (such as $\doteq$) from meta-level operators (such as $=$). Java's `==` operator translates to $\doteq$ in the logic. The identifier `res` refers to the method's return value.

In class `ArrayList`, the meaning of the symbol `size` is defined by the method body for `size`. Similarly, we need to give a definition for the abstract field *inv*, which we do with the following axiom:

$$\begin{aligned} &\textit{exactInstance}_{\texttt{ArrayList}}(\texttt{this}) \\ &\rightarrow \big(\texttt{this}.\textit{inv} \;\leftrightarrow\; \texttt{this.a} \doteq\!\!\!\neq \texttt{null} \land \texttt{this.n} < \texttt{this.a.length} \\ &\qquad\qquad \land\; \forall \textit{Int}\, i; (0 \leq i \land i < \texttt{this.n} \rightarrow \texttt{this.a[}i\texttt{]} \doteq\!\!\!\neq \texttt{null})\big) \end{aligned} \tag{1}$$

For a type A and an expression `e`, the formula $\textit{exactInstance}_A(\texttt{e})$ evaluates to true in a state if the dynamic type of `e` is A. Intuitively, *inv* represents an "object invariant" for `List`, i.e., a consistency property on its objects, where the exact nature of this property is defined privately in subclasses of the interface. With the definition for `ArrayList` in (1), the implementation of `get` in `ArrayList` satisfies the method contract for `get`.

For method `m` in `Client`, we give the following method contract:

$$\textit{pre}\colon \texttt{l} \doteq\!\!\!\neq \texttt{null} \land \texttt{l}.\textit{inv} \land 0 < \texttt{l.size()} \qquad \textit{post}\colon \texttt{res} \doteq\!\!\!\neq \texttt{null}$$

Can we verify that `m` complies with this contract, provided that all implementations of `get` satisfy the contract for `get`? Unfortunately, the answer is no. The problem is that even though the precondition guarantees properties about the *initial* values of `l.`*inv* and `l.size()`, this does not imply that these properties still hold when `get` is called at the end of `m`, because of the intervening change to `x`. This is an instance of a general problem when using data abstraction in specifications [8, Challenge 3]: without further measures, any change to the heap can affect the value of an abstract field or of a method in an unknown way.

As a solution, we introduce *dependency contracts* (also known as *depends clauses* [9]) into our specifications. A dependency contract restricts the set of

memory locations that are allowed to influence the value of an abstract field or of a method, provided that some precondition holds. An example for a correct dependency contract for method `size` in `ArrayList` is one which states that the method result is allowed to depend only on $\{(\texttt{this}, \texttt{n})\}$, where the expression $\{(\texttt{this}, \texttt{n})\}$ refers to the set consisting of the single memory location given by the field `n` for the object represented by the expression `this`.

How can we express a useful dependency contract for *inv* or `size` in `List`, even though here we do not have access to the locations implementing the list? We see that the need for data abstraction also extends to location sets. Our solution is to use *dynamic frames* [7], i.e., abstract fields that evaluate to sets of memory locations. For the specification of `List`, we declare a dynamic frame *locs*. In `ArrayList`, we define *locs* via the following axiom:

$$\mathit{exactInstance}_{\texttt{ArrayList}}(\texttt{this}) \;\rightarrow\; \texttt{this}.\mathit{locs} \doteq (\texttt{this.*} \dot{\cup} \texttt{this.a.*}) \qquad (2)$$

The expression `o.*` refers to the set of all fields of the object represented by the expression `o`. If `o` has an array type, then `o.*` denotes all components of the array.

We use the dynamic frame *locs* to give dependency contracts for both *inv* and `size`: both are supposed to depend at most on the locations in *locs*. These dependency contracts are satisfied in `ArrayList`, because both $\texttt{this}.\mathit{inv}$ (as defined by (1)) and `this.size()` (as defined by the method body in Fig. 1) read only locations that are members of $\texttt{this}.\mathit{locs}$ as defined by (2).

Finally, we modify the precondition of `m` in `Client` to be as follows:

$$\mathit{pre}\colon \texttt{l} \neq \texttt{null} \wedge \texttt{l}.\mathit{inv} \wedge 0 < \texttt{l.size()} \wedge (\texttt{this}, \texttt{x}) \dot{\notin} \texttt{l}.\mathit{locs}$$

Now, when reasoning about the correctness of `m`, we know that the location $(\texttt{this}, \texttt{x})$ is not a member of the (unknown) set of locations $\texttt{l}.\mathit{locs}$ on which $\texttt{l}.\mathit{inv}$ and `l.size()` may depend. Thus, changing the value of this location cannot have an effect on the values of $\texttt{l}.\mathit{inv}$ and `l.size()`, and so $\texttt{l}.\mathit{inv} \wedge 0 < \texttt{l.size()}$ must still be true when method `get` is called at the end of `m`. Together with the method contract for `get`, this guarantees that the return value of `get` is different from `null`, and thus that the postcondition of `m` is satisfied.

In general, we also need *modifies clauses* in method contracts, which fix a set of locations that may at most be modified by a method, provided that the precondition of the contract holds upon method entry. In the example, `get` and `size` are supposed to not have side effects, so we can use modifies clauses of $\dot{\emptyset}$ (an empty set of locations). For `add`, $\texttt{this}.\mathit{locs}$ can serve as a modifies clause.

Also, as the value of the dynamic frame $\texttt{this}.\mathit{locs}$ is itself state-dependent, specifications of the behaviour of *locs* itself are needed in order to make the specification fully useful for modular verification. We can give a dependency contract for $\texttt{this}.\mathit{locs}$ stating that its value depends at most on the locations in $\texttt{this}.\mathit{locs}$ itself; this is satisfied by the definition (2), because the only location it reads is $(\texttt{this}, \texttt{a})$, which itself is defined to be a member of $\texttt{this}.\mathit{locs}$. We may also want to specify (via method contracts) that after the construction of an `ArrayList` object, the set $\texttt{this}.\mathit{locs}$ contains only freshly allocated locations, and that method `add` can add to the set only freshly allocated locations (the latter is sometimes called the "swinging pivots property" [11, 7]).

3 Java Dynamic Logic With an Explicit Heap Model

In this section, we present a dynamic logic and a sequent calculus for the modular verification of Java source code wrt. dynamic frames style specifications. It is a variation of the dynamic logic underlying the KeY verification tool [2]. The main difference is in the logical modelling of heap memory. For complete formal definitions please see the technical report [19], which accompanies this paper.

3.1 Syntax and Semantics

The *syntax* of the logic is based on a *signature* Σ, which comprises a set $\mathcal{T}$ of *types*, a partial order $\sqsubseteq$ called the *subtype relation*, and disjoint sets of (logical) *variables* $\mathcal{V}$, *program variables* $\mathcal{PV}$, *function symbols* $\mathcal{F}$, and *predicate symbols* $\mathcal{P}$. All variables and symbols are typed. We use the notation $x : A$ to indicate that the type of x is A, the notation $f : A_1, \ldots, A_n \to B$ to indicate that the function symbol f maps arguments of types $A_1, \ldots, A_n$ to type B, and the notation $p : A_1, \ldots, A_n$ to indicate that the predicate symbol p represents a relation on the types $A_1, \ldots, A_n$. The signature Σ is specific to a Java program to be verified. All types of this program also appear as types in $\mathcal{T}$, and all local variables appear as program variables in $\mathcal{PV}$. In contrast to program variables, *logical* variables may not appear in programs, but may be quantified. The type $Any \in \mathcal{T}$ is a supertype of all types of the program.

The set Fma_Σ of *formulas* and the set Trm_Σ of *terms* are defined mostly as in classical typed first-order logic. For any type $A \in \mathcal{T}$, we have the set $Trm^A_\Sigma \subseteq Trm_\Sigma$ of terms of type A. In addition to the operators of first-order logic, Java dynamic logic includes modal operators $[\mathtt{p}]$ and $\langle \mathtt{p} \rangle$ for every executable Java program fragment $\mathtt{p}$. If $\varphi \in Fma_\Sigma$ is a formula, then both $[\mathtt{p}]\varphi$ and $\langle \mathtt{p} \rangle \varphi$ are also formulas. Our version of dynamic logic also includes another kind of modal operator, called *updates* [18]. An update is denoted as $\mathtt{a}_1 := t_1 \parallel \ldots \parallel \mathtt{a}_n := t_n$, where $\mathtt{a}_1, \ldots, \mathtt{a}_n \in \mathcal{PV}$, and where $t_1, \ldots, t_n$ are terms such that the type of t_i is a subtype of the type of $\mathtt{a}_i$. The set of updates is called Upd_Σ. If u is an update and t is a term or formula, then $\{u\}t$ is also a term or formula, respectively.

The *semantics* of a term or formula is given by an *interpretation* which maps all function symbols to functions and all predicate symbols to relations, and by a *state* which maps all program variables to values. First-order terms and formulas are evaluated as usual. The formula $[\mathtt{p}]\varphi$ holds in a state s if the execution of $\mathtt{p}$ started in s either does not terminate, or terminates in a state s' such that φ holds in s' (*partial* correctness). The formula $\langle \mathtt{p} \rangle \varphi$ holds if $[\mathtt{p}]\varphi$ holds, and if additionally $\mathtt{p}$ does indeed terminate (*total* correctness). Like a program $\mathtt{p}$, an update u changes the state: executing the update $\mathtt{a}_1 := t_1 \parallel \ldots \parallel \mathtt{a}_n := t_n$ in a state s leads to an updated state s' which is identical to s, except that the program variables $\mathtt{a}_i$ have been assigned the values of the terms t_i in parallel. Evaluating $\{u\}t$ in s is the same as evaluating t in the updated state s'. A formula is called *logically valid* if it holds for all interpretations and all states.

3.2 Sequent Calculus

The calculus we use to reason about logical validity of formulas is a *sequent calculus*. A proof in the sequent calculus is a tree of so-called *sequents* $\Gamma \Rightarrow \Delta$, in which Γ (called the *antecedent*) and Δ (the *succedent*) are finite sets of formulas. A sequent $\Gamma \Rightarrow \Delta$ has the same semantic truth value as the formula $\bigwedge \Gamma \rightarrow \bigvee \Delta$.

An *inference rule* of the sequent calculus has a number of sequents as its *premisses* and a single sequent as its *conclusion*; it is *sound* if logical validity of all premisses implies logical validity of the conclusion. In addition to inference rules, our calculus contains *rewrite rules*, which allow rewriting a term or formula at an arbitrary position in a sequent. A rewrite rule is sound if the original and the rewritten term or formula are equal resp. logically equivalent. We formulate both sequent and rewrite rules schematically to achieve a finite representation of the calculus. For example, in the following two (sound) rule schemata, the *schema formulas* φ and ψ can be instantiated with arbitrary formulas, and Γ and Δ with arbitrary sets of formulas:

$$\text{(andRight)}\quad \frac{\Gamma \Rightarrow \varphi, \Delta \quad \Gamma \Rightarrow \psi, \Delta}{\Gamma \Rightarrow \varphi \wedge \psi, \Delta} \qquad \text{(andIdem)}\quad \psi \wedge \psi \;\rightsquigarrow\; \psi$$

Starting with the sequent to prove as root, a *proof tree* is constructed by applying sequent and rewrite rules. For the application of a sequent rule to a leaf in the proof tree, this sequent must be identical to the conclusion of the rule. The rule's premisses are then added as new children to the former leaf. A rewrite rule $t_1 \rightsquigarrow t_2$ can be applied to a leaf by replacing one occurrence of t_1 in its sequent by t_2. Provided that all applied rules are sound, it is guaranteed that at any time during this process, validity of all the leaves implies validity of the root sequent. If one arrives at a tree whose leaves are all obviously valid, one has proven the validity of the original proof obligation.

3.3 Heap Model

In contrast to [2, 18], where the Java heap is modelled via a *non-rigid* function symbol $\mathtt{f} : A \rightarrow B$ for every Java field `f` of type `B` declared in class `A`, here we follow [17, 22, 1, 21] in modelling the heap using the *theory of arrays* [13]. The fields of our Java program are represented as constant symbols of a type $\mathit{Field} \in \mathcal{T}$, which are axiomatised to have distinct values. Heaps now occur "explicitly" in formulas, as terms of a type $\mathit{Heap} \in \mathcal{T}$. The values of this type are arrays indexed by locations, i.e., by pairs of (Object, Field) values. Reading from and writing to a heap is done with the help of the function symbols $\mathit{select}_A : \mathit{Heap}, \mathit{Object}, \mathit{Field} \rightarrow A$ and $\mathit{store} : \mathit{Heap}, \mathit{Object}, \mathit{Field}, \mathit{Any} \rightarrow \mathit{Heap}$. These are standard, except that for convenience we use a separate symbol select_A for every type $A \in \mathcal{T}$, which implicitly casts the retrieved value to a desired type A. A global program variable $\mathtt{heap} : \mathit{Heap} \in \mathcal{PV}$ holds the current heap of the program. We will in the following often use the more concise notation $o.f$ instead of $\mathit{select}_A(\mathtt{heap}, o, f)$.

The axiom of the theory of arrays manifests itself in the rewrite rule selectOf-Store depicted in Fig. 2: The value $\mathit{select}_A(\mathit{store}(h, o, f, t), o', f')$ of a location

$$select_A(store(h,o,f,t),o',f') \rightsquigarrow \quad (\mathsf{selectOfStore})$$
$$if(o \doteq o' \wedge f \doteq f')then(cast_A(t))else(select_A(h,o',f'))$$

$$select_A(anon(h,s,h'),o,f) \rightsquigarrow \quad (\mathsf{selectOfAnon})$$
$$if\big(((o,f) \mathrel{\dot\in} s \wedge f \not\doteq created) \vee (o,f) \mathrel{\dot\in} freshLocs(h)\big)$$
$$then(select_A(h',o,f))$$
$$else(select_A(h,o,f))$$

$$cast_A(t) \rightsquigarrow t \qquad \text{for } t \in Trm_\Sigma^{A'} \text{ and } A' \sqsubseteq A \quad (\mathsf{cast})$$

$$(o,f) \mathrel{\dot\in} freshLocs(h) \rightsquigarrow \quad (\mathsf{inFreshLocs})$$
$$o \not\doteq \texttt{null} \wedge select_{Boolean}(h,o,created) \doteq FALSE$$

$$[\texttt{a = } t\texttt{; } \ldots]\varphi \rightsquigarrow \{\mathsf{a} := t\}[\ldots]\varphi \quad (\mathsf{assignLocal})$$

$$[o\texttt{.}f\texttt{ = } t\texttt{; } \ldots]\varphi \rightsquigarrow \{\texttt{heap} := store(\texttt{heap},o,f,t)\}[\ldots]\varphi \quad (\mathsf{assignField})$$

Fig. 2. A selection of rewrite rules for heap modifications and location sets

(o,f) retrieved from a modified heap $store(h,o,f,t)$ depends on whether the retrieved location is the previously modified one, i.e., whether $(o',f') \doteq (o,f)$ holds. If so, the assigned value t is read, otherwise the retrieval is delegated to the embedded heap h as $select_A(h,o',f')$. The type coercion operation $cast_A(t)$ can later be removed using the rule cast if the heap has been used consistently.

In our logic, all states share a common semantic domain (this is known as the *constant domain assumption*). Therefore, we need a means to explicitly distinguish between already-created and not-yet-created objects in the sense of Java. We use an implicit ("ghost") field $created : Field$ for this purpose: we consider an object o to be created in a state if and only if $o\texttt{.}created$ evaluates to true in this state. Allocating an object via Java's `new` operator implicitly sets its *created* field to true.

Dynamic frames are supported via a type $LocSet \in \mathcal{T}$. Terms of type *LocSet* evaluate to sets of memory locations. Our signatures contain the symbols $\dot\emptyset$, $\dot\cup$, $\dot\cap$, $\dot\setminus$, $\mathrel{\dot\in}$, $\dot\subseteq$, *disjoint* and *allLocs*, which are pre-defined to have their expected set-theoretical semantics. The function symbol $freshLocs : Heap \to LocSet$ yields for every heap the set of locations (o,f) for which the object o is not yet created in this heap. The corresponding rule inFreshLocs is shown in Fig. 2.

When dispatching a method call in a proof with the help of a contract for the called method (Sect. 4), we use a special heap modification function $anon : Heap, LocSet, Heap \to Heap$. Roughly, the heap $anon(h,s,h')$ is identical to h' in the locations of s, and it is identical to the "original" heap h in all other locations. The exact semantics of *anon* is described by the rewrite rule selectOfAnon in Fig. 2: independently of the set s, going from h to $anon(h,s,h')$ for some unknown h' (a process which we call an "anonymisation" of the heap h wrt. the set s) never leads to deallocating existing objects, but always implicitly allows for the allocation of new objects. This resembles the behaviour of method calls in Java.

We also introduce a unary predicate symbol $\mathit{wellFormed} : \mathit{Heap}$, which can be axiomatised as

$$\forall \mathit{Heap}\, h; \big(\mathit{wellFormed}(h) \leftrightarrow \forall \mathit{Object}\, o, p; \forall \mathit{Field}\, f; \\ (\mathit{select}_{\mathit{Any}}(h, o, f) \doteq p \rightarrow (p \doteq \texttt{null} \vee \mathit{select}_{\mathit{Boolean}}(h, p, \mathit{created}) \doteq \mathit{TRUE}))\big)\ ,$$

i.e., a heap h is considered well-formed if any object p which is referenced by some location (o, f) is either the `null` object or an object which has already been created. The semantics of Java guarantees that $\mathit{wellFormed}(\texttt{heap})$ holds for all states occurring during the execution of a Java program.

3.4 Symbolic Execution

A central component of our calculus is a set of rule schemata that allow us to transform formulas with program modalities and updates into formulas without. This process is called *symbolic execution*. Programs are systematically processed in a forward manner: whenever we encounter a formula $[\texttt{p;q}]\varphi$, we handle the statement `p` first, and leave the formula $[\texttt{q}]\varphi$ to be treated later. This forward treatment of programs is based on the concept of updates. There is also a set of rules which handle the simplification and application of updates to terms and formulas. The theory of rule-based update treatment has been elaborated in [18].

Two rules for symbolic execution, namely assignLocal and assignField, are shown in Fig. 2. The corresponding rules for the modality $\langle\cdot\rangle$ read accordingly. Both rules are used to execute assignment statements, either for a local variable `a` or for a field reference $o.f$. Let t be a side-effect free Java expression which (after some syntactic adaptions like `==` to $\doteq$, `&&` to $\wedge$, etc.) can be read as a term in our logic. An assignment statement `a = t;` which assigns to `a` the value of the expression t, describes then the same state modification as the update $\texttt{a} := t$. This is captured in the symbolic execution rule assignLocal. An assignment to a location $o.f$ is treated differently: it corresponds to a modification of the global program variable `heap`. We do not show the rules for other language features here, as they are numerous and largely orthogonal to the focus of this paper. We also ignore Java exceptions throughout the paper, which allows for a more readable presentation of rules and proof obligations. For a more complete treatment of Java language features, please refer to [2].

Fig. 3 depicts a small example proof. Therein, $\texttt{o} \in \mathcal{PV}$ is a local variable of a reference type, $\texttt{f} : \mathit{Field} \in \mathcal{F}$ is a constant symbol, and $\texttt{a} : \mathit{Int} \in \mathcal{PV}$ is a local variable. Symbolic execution first converts the two Java assignments into corresponding updates. The updates are then simplified into a single update that performs both state changes in parallel. The left sub-update $\texttt{heap} := \mathit{store}(\texttt{heap}, \texttt{o}, \texttt{f})$ can be simplified away, because the variable `heap` does not occur in the scope of the update any more, and thus its value is irrelevant. The rule selectOfStore is applied inside the remaining update, followed by an obvious simplification of the resulting *if-then-else*-term. The type cast operator can be removed with the cast rule, because 0 is of type *Int*. Finally, the update is applied to the sub-formula $\texttt{a} \doteq 0$ as a substitution, resulting in an obviously valid formula. Hence, we have proven that the original formula is valid as well.

$$
\begin{array}{ll}
 & [\texttt{o.f = 0; a = o.f;}](\texttt{a} \doteq 0) \\
\overset{\text{assignField}}{\rightsquigarrow} & \{\texttt{heap} := \mathit{store}(\texttt{heap}, \texttt{o}, \texttt{f}, 0)\}[\texttt{a = o.f;}](\texttt{a} \doteq 0) \\
\overset{\text{assignLocal}}{\rightsquigarrow} & \{\texttt{heap} := \mathit{store}(\texttt{heap}, \texttt{o}, \texttt{f}, 0)\}\{\texttt{a} := \mathit{select}_{\mathit{Int}}(\texttt{heap}, \texttt{o}, \texttt{f})\}(\texttt{a} \doteq 0) \\
\overset{\mathit{upd.\ simpl.}}{\rightsquigarrow} & \{\texttt{heap} := \mathit{store}(\texttt{heap}, \texttt{o}, \texttt{f}, 0) \parallel \texttt{a} := \mathit{select}_{\mathit{Int}}(\mathit{store}(\texttt{heap}, \texttt{o}, \texttt{f}, 0), \texttt{o}, \texttt{f})\}(\texttt{a} \doteq 0) \\
\overset{\mathit{upd.\ simpl.}}{\rightsquigarrow} & \{\texttt{a} := \mathit{select}_{\mathit{Int}}(\mathit{store}(\texttt{heap}, \texttt{o}, \texttt{f}, 0), \texttt{o}, \texttt{f})\}(\texttt{a} \doteq 0) \\
\overset{\text{selectOfStore}}{\rightsquigarrow} & \{\texttt{a} := \mathit{if}(\texttt{o} \doteq \texttt{o} \wedge \texttt{f} \doteq \texttt{f})\mathit{then}(\mathit{cast}_{\mathit{Int}}(0))\mathit{else}(\mathit{select}_{\mathit{Int}}(\texttt{heap}, \texttt{o}, \texttt{f}))\}(\texttt{a} \doteq 0) \\
\overset{\mathit{simpl.}}{\rightsquigarrow} & \{\texttt{a} := \mathit{cast}_{\mathit{Int}}(0)\}(\texttt{a} \doteq 0) \\
\overset{\text{cast}}{\rightsquigarrow} & \{\texttt{a} := 0\}(\texttt{a} \doteq 0) \\
\overset{\mathit{upd.\ appl.}}{\rightsquigarrow} & 0 \doteq 0
\end{array}
$$

Fig. 3. Example proof

4 Contracts and Proof Obligations

Both abstract fields, such as *inv* and *locs* in Sect. 2, and side-effect free methods such as `size` are represented in the logic as so-called *observer symbols*.

Definition 1 (Observer symbols). *An* observer symbol for type A with argument types $B_1, \ldots, B_n$ *is either a function symbol* $\mathit{obs} : \mathit{Heap}, A, B_1, \ldots, B_n \to B \in \mathcal{F}$ *or a predicate symbol* $\mathit{obs} : \mathit{Heap}, A, B_1, \ldots, B_n \in \mathcal{P}$*, where* $A \sqsubseteq \mathit{Object}$*.*

As syntactic sugar, we sometimes write $o.\mathit{obs}(p_1, \ldots, p_n)$ to denote the term or formula $\mathit{obs}(\texttt{heap}, o, p_1, \ldots, p_n)$. This (deliberately) resembles the notation $o.f$ for field access terms $\mathit{select}_A(\texttt{heap}, o, f)$. Nevertheless, an observer symbol does not give rise to a memory location; instead, it "observes" (i.e., it depends on) the values of memory locations. For an observer symbol `m` representing a side-effect free method without parameters, we sometimes write `o.m()` instead of `o.m`.

We have seen in Sect. 2 that the value of abstract fields is defined via axioms such as (1) and (2). Similarly, observer symbols representing *methods* are defined via axioms such as the following (where `this` and `r` are fresh program variables):

$$
\begin{aligned}
&\mathit{exactInstance}_{\texttt{ArrayList}}(\texttt{this}) \\
&\to \forall \mathit{Int}\, i; \big(\texttt{this.size()} \doteq i \;\leftrightarrow\; \langle \texttt{r = this.size();} \rangle \texttt{r} \doteq i\big)
\end{aligned}
\tag{3}
$$

The axiom uses the modal operator $\langle\cdot\rangle$ to connect the observer symbol `size` with a call to method `size` in class `ArrayList`.

Axioms (1), (2), (3) are supposed to hold for all values of the program variables `this` and `heap`. The corresponding universally quantified versions of the axioms can be used as assumptions in proofs for the correctness of `ArrayList`. We could also allow using them in other proofs, but this is undesirable for reasons of modularity: the axioms are implementational secrets of `ArrayList`, and should not be exposed to other classes.

Besides observer symbols and axioms, a *specification* in our setting consists of a set of *method contracts* constraining the behaviour of methods, and of a

set of *dependency contracts* constraining the dependencies of observer symbols. Both kinds of contract give rise to *proof obligations*, i.e., formulas whose validity must be proven in order for the program to be considered correct. On the other hand, both kinds of contract can also be used as assumptions in the proofs of other contracts, via special *rules*. Subsect. 4.1 defines method contracts, the corresponding proof obligation, and the corresponding rule; Subsect. 4.2 does the same for dependency contracts. Note that for simplicity of presentation, we omit the treatment of void methods, static methods, static fields, and constructors.

4.1 Method Contracts

Definition 2 (Method contracts). *A* method contract *mct is a tuple*

$$mct = \big(\mathtt{m}, \mathtt{this}, (\mathtt{p}_1, \ldots, \mathtt{p}_n), \mathtt{res}, \mathtt{hPre}, pre, post, mod, \tau\big)$$

where $\mathtt{m}$ *is a Java method; where* $\mathtt{this} : A \in \mathcal{PV}$ *such that* $\mathtt{m}$ *is defined for receiver objects of type A; where* $\mathtt{p}_1, \ldots, \mathtt{p}_n, \mathtt{res} \in \mathcal{PV}$ *such that their types correspond to the declared signature of* $\mathtt{m}$*; where* $\mathtt{hPre} : Heap \in \mathcal{PV}$*; and where* $pre, post \in Fma_\Sigma$*,* $mod \in Trm_\Sigma^{LocSet}$*, and* $\tau \in \{partial, total\}$*.*

The program variables $\mathtt{this}$ and $\mathtt{p}_1, \ldots, \mathtt{p}_n$ may be used in the precondition *pre*, in the postcondition *post* and in the modifies clause *mod* to represent the receiver object of $\mathtt{m}$ and the arguments to $\mathtt{m}$, respectively. The variables $\mathtt{res}$ and $\mathtt{hPre}$ can be used in *post* to refer to the method's return value and to the value of $\mathtt{heap}$ in the pre-state. The "termination marker" τ indicates whether the contract demands partial or total correctness.

Definition 3 (Proof obligation for method contracts). *Given a method contract* $mct = \big(\mathtt{m}, \mathtt{this}, (\mathtt{p}_1, \ldots, \mathtt{p}_n), \mathtt{res}, \mathtt{hPre}, pre, post, mod, \tau\big)$ *with* $\mathtt{this} : A$*, and given a type* $B \sqsubseteq A$*, the proof obligation* $CorrectMethodContract(mct, B) \in Fma_\Sigma$ *is defined as*

$$\begin{aligned} &pre \wedge reachableState \wedge exactInstance_B(\mathtt{this}) \\ &\rightarrow \{\mathtt{hPre} := \mathtt{heap}\}[\![\mathtt{res = this.m(p}_1, \ldots, \mathtt{p}_n\mathtt{);}]\!](post \wedge frame), \end{aligned}$$

where $[\![\cdot]\!]$ *stands for* $[\cdot]$ *if* $\tau = partial$ *and for* $\langle\cdot\rangle$ *if* $\tau = total$*, and where*

- *reachableState is the formula*

$$\begin{aligned} &wellFormed(\mathtt{heap}) \wedge \mathtt{this} \neq \mathtt{null} \wedge \mathtt{this}.created \doteq TRUE \\ &\wedge \bigwedge_{i \in \{1,\ldots,n\},\ \mathtt{p}_i : A \text{ for some } A \sqsubseteq Object} (\mathtt{p}_i \doteq \mathtt{null} \vee \mathtt{p}_i.created \doteq TRUE) \end{aligned}$$

- *frame is the formula*

$$\begin{aligned} \forall Object\, o; \forall Field\, f; \big((o, f) \mathrel{\dot\in} \{\mathtt{heap} := \mathtt{hPre}\}\big(mod \mathbin{\dot\cup} freshLocs(\mathtt{heap})\big) \\ \vee\, o.f \doteq \{\mathtt{heap} := \mathtt{hPre}\}o.f\big) \end{aligned}$$

The *reachableState* property is guaranteed by Java itself: the heap is well-formed, the receiver object is created, and all objects passed as arguments are either `null` or created. The formula *frame* is the frame condition generated from the modifies clause *mod*: after executing `m`, only locations in *mod* (interpreted in the pre-state) and "fresh" locations may have changed compared to the pre-state.

For method `get` with *pre* and *post* from Sect. 2, $\tau = total$, and $B =$ `ArrayList`, we get the following instance of *CorrectMethodContract*:

$$\begin{aligned}&\texttt{this}.\mathit{inv} \land 0 \leq \texttt{i} \land \texttt{i} < \texttt{this.size()} \land \mathit{wellFormed}(\texttt{heap})\\ &\land \texttt{this} \neq \texttt{null} \land \texttt{this}.\mathit{created} \doteq \mathit{TRUE} \land \mathit{exactInstance}_{\texttt{ArrayList}}(\texttt{this})\\ &\rightarrow \{\texttt{hPre} := \texttt{heap}\}\langle\texttt{res = this.get(i);}\rangle(\texttt{res} \neq \texttt{null} \land \mathit{frame})\end{aligned}$$

where *frame* with a modifies clause $mod = \dot{\emptyset}$ states that only fresh locations may have been changed by `m`. The formula is valid under the assumption of (the universally quantified versions of) axioms (1) and (3). When proving this, one of the first steps is to inline the body of method `get`, which is possible because we know the exact type of `this` and, hence, do not have to consider dynamic dispatch.

The following rule allows using a method contract as an assumption:

Definition 4 (Rule useMethodContract).

$$\frac{\begin{array}{l}\Gamma \Rightarrow \{u\}\{w\}(\mathit{pre} \land \mathit{reachableState}),\ \Delta\\ \Gamma \Rightarrow \{u\}\{w\}\{\texttt{hPre} := \texttt{heap}\}\{v\}(\mathit{post} \land \mathit{reachableState}' \rightarrow [\![\ldots]\!]\varphi),\ \Delta\end{array}}{\Gamma \Rightarrow \{u\}[\![\texttt{r = o.m(p}'_1,\ldots,\texttt{p}'_n\texttt{); } \ldots]\!]\varphi,\ \Delta}$$

where:

- $\texttt{o} \in \mathit{Trm}^A_\Sigma$ *for some* $A \in \mathcal{T}$ *such that there is a method contract*

$$\mathit{mct} = (\texttt{m}, \texttt{this}, (\texttt{p}_1,\ldots,\texttt{p}_n), \texttt{res}, \texttt{hPre}, \mathit{pre}, \mathit{post}, \mathit{mod}, \tau)$$

  where $\texttt{this}: A$*; where* $\tau = total$ *if the modality* $[\![\cdot]\!]$ *is* $\langle\cdot\rangle$*, and where* τ *does not matter otherwise; and where* $\texttt{this}, \texttt{p}_1,\ldots,\texttt{p}_n, \texttt{res}$ *and* `hPre` *do not occur in the formula* $[\![\texttt{r = o.m(p}'_1,\ldots,\texttt{p}'_n\texttt{); } \ldots]\!]\varphi$
- $\texttt{p}'_1,\ldots,\texttt{p}'_n$ *are terms*
- *reachableState* $\in \mathit{Fma}_\Sigma$ *is as in Def. 3, and reachableState*$'$ *is the formula*

$$\mathit{wellFormed}(\texttt{heap}) \land (\texttt{res} \doteq \texttt{null} \lor \texttt{res}.\mathit{created} \doteq \mathit{TRUE})$$

  if $\texttt{res}: B$ *for some* $B \sqsubseteq \mathit{Object}$*, and the formula wellFormed*(`heap`) *otherwise*
- $v = \big(\texttt{heap} := \mathit{anon}(\texttt{heap}, \mathit{mod}, h) \parallel \texttt{r} := r' \parallel \texttt{res} := r'\big)$
- $w = \big(\texttt{this} := \texttt{o} \parallel \texttt{p}_1 := \texttt{p}'_1 \parallel \ldots \parallel \texttt{p}_n := \texttt{p}'_n\big)$
- $h: \mathit{Heap} \in \mathcal{F}$ *and* $r' \in \mathcal{F}$ *are fresh symbols, i.e., they do not yet occur anywhere in the proof when applying the rule*

Like *reachableState*, *reachableState*$'$ is a property guaranteed by Java. The update v "anonymises" the locations that may be changed by the call to `m`, namely

the members of the modifies clause *mod*, by setting them to unknown values with the help of the new symbol h. It also sets the result variable `r`, and its counterpart `res`, to an unknown value r'. The update w instantiates the variables used in the contract with the corresponding terms in the method call.

Instead of using *anon*, we could also anonymise (or "havoc" [10]) the entire heap, and use a framing formula like *frame* in Def. 3 to express that some locations do *not* change. The advantage of our approach is that it avoids the universal quantifiers of *frame* in applications of useMethodContract.

The useMethodContract rule is sound, provided that for all subtypes $B \sqsubseteq A$ of the static receiver type A, the proof obligation $\mathit{CorrectMethodContract}(\mathit{mct}, B)$ is logically valid. A proof of this theorem is contained in [19]. We forbid "circular" applications of the rule, such as applying the rule on a call to the method which is itself being verified in the current proof. An extension to support recursion is possible, but beyond the scope of this paper.

4.2 Dependency Contracts

Definition 5 (Dependency contracts). *A* dependency contract *is a tuple*

$$\mathit{depct} = (\mathit{obs}, \mathtt{this}, (\mathtt{p}_1, \ldots, \mathtt{p}_n), \mathit{pre}, \mathit{dep})$$

where obs is an observer symbol for type A' with argument sorts $B_1, \ldots, B_n$; where $\mathtt{this} : A \in \mathcal{PV}$ such that $A \sqsubseteq A'$; where $\mathtt{p}_1 : B_1, \ldots, \mathtt{p}_n : B_n \in \mathcal{PV}$; and where $\mathit{pre} \in \mathit{Fma}_\Sigma$, $\mathit{dep} \in \mathit{Trm}_\Sigma^{\mathit{LocSet}}$.

The program variables `this` and $\mathtt{p}_1, \ldots, \mathtt{p}_n$ can be used in the precondition *pre* and the depends clause *dep* to stand for the receiver object and the parameters of *obs*, respectively. An example for a dependency contract in the context of the program of Sect. 2 is $(\mathit{inv}, \mathtt{this}, (), \mathtt{this}.\mathit{inv}, \mathtt{this}.\mathit{locs})$, which demands that the value of $\mathtt{this}.\mathit{inv}$ should depend only on locations in $\mathtt{this}.\mathit{locs}$, provided that $\mathtt{this}.\mathit{inv}$ is true at the time.

Definition 6 (Proof obligation for dependency contracts). *For a dependency contract $\mathit{depct} = (\mathit{obs}, \mathtt{this}, (\mathtt{p}_1, \ldots, \mathtt{p}_n), \mathit{pre}, \mathit{dep})$ with $\mathtt{this} : A$, and for a type $B \sqsubseteq A$, the proof obligation $\mathit{CorrectDependencyContract}(\mathit{depct}, B) \in \mathit{Fma}_\Sigma$ is defined as follows:*

$$\begin{aligned} &\mathit{pre} \wedge \mathit{reachableState} \wedge \mathit{exactInstance}_B(\mathtt{this}) \\ &\rightarrow \mathtt{this}.\mathit{obs}(\mathtt{p}_1, \ldots, \mathtt{p}_n) \\ &\quad \equiv \{\mathtt{heap} := \mathit{anon}(\mathtt{heap}, \mathit{allLocs} \setminus \mathit{dep}, h)\}\big(\mathtt{this}.\mathit{obs}(\mathtt{p}_1, \ldots, \mathtt{p}_n)\big) \end{aligned}$$

where $\mathit{reachableState} \in \mathit{Fma}_\Sigma$ is as in Def. 3, where $h : \mathit{Heap} \in \mathcal{F}$ is fresh, and where $\equiv$ stands for $\doteq$ if $\mathit{obs} \in \mathcal{F}$ and for $\leftrightarrow$ if $\mathit{obs} \in \mathcal{P}$.

The proof obligation formalises the notion of *obs* "depending" only on the locations in *dep*: if we change all locations except for *dep* in an unknown way, then

this must not affect *obs*. For the dependency contract for *inv* above, and for $B = \texttt{ArrayList}$, we get the following instance of *CorrectDependencyContract*:

$$\begin{aligned}&\texttt{this}.\mathit{inv} \wedge \mathit{wellFormed}(\texttt{heap}) \wedge \texttt{this} \neq \texttt{null} \wedge \texttt{this}.\mathit{created} \doteq \mathit{TRUE}\\ &\wedge \mathit{exactInstance}_{\texttt{ArrayList}}(\texttt{this})\\ &\rightarrow \big(\texttt{this}.\mathit{inv} \leftrightarrow \{\texttt{heap} := \mathit{anon}(\texttt{heap}, \mathit{allLocs} \setminus \texttt{this}.\mathit{locs}, h)\}(\texttt{this}.\mathit{inv})\big)\end{aligned}$$

The formula is valid under the assumption of axioms (1) and (2), because all locations read by (1) are defined to be a part of $\texttt{this}.\mathit{locs}$ by (2). Analogously, (3) defines `this.size()` such that it also depends only on the locations in $\texttt{this}.\mathit{locs}$ as defined by (2).

Definition 7 (Rule useDependencyContract).

$$\frac{\Gamma,\ \mathit{guard} \rightarrow \mathit{equal}\ \Rightarrow\ \Delta}{\Gamma\ \Rightarrow\ \Delta}$$

where:

- *the term or formula* $\mathit{obs}(h^{new}, \texttt{o}, \texttt{p}'_1, \ldots, \texttt{p}'_n)$ *occurs in* Γ *or* Δ*, where* $h^{new} = f_1(f_2(\ldots(f_m(h^{base}, \ldots))))$ *with* $f_1, \ldots, f_m \in \{\mathit{store}, \mathit{anon}\}$*,* $h^{base} \in \mathit{Trm}^{Heap}_{\Sigma}$
- $\texttt{o} \in \mathit{Trm}^{A}_{\Sigma}$ *for some* $A \in \mathcal{T}$ *such that there is a dependency contract* $\mathit{depct} = (\mathit{obs}, \texttt{this}, (\texttt{p}_1, \ldots, \texttt{p}_n), \mathit{pre}, \mathit{dep})$*, where* $\texttt{this} : A$*, and where both* `this` *and* $\texttt{p}_1, \ldots, \texttt{p}_n$ *do not occur in* Γ *or* Δ
- $\texttt{hPre} : \mathit{Heap} \in \mathcal{PV}$ *is fresh,* $\mathit{mod} = \mathit{allLocs} \setminus \mathit{dep}$
- $\mathit{reachableState}, \mathit{frame} \in \mathit{Fma}_{\Sigma}$ *are as in Def. 3,* $w \in \mathit{Upd}_{\Sigma}$ *is as in Def. 4*
- $\mathit{noDeallocs} \in \mathit{Fma}_{\Sigma}$ *is the formula*

$$\begin{aligned}&\mathit{freshLocs}(\texttt{heap}) \subseteq \mathit{freshLocs}(\texttt{hPre})\\ &\wedge\ \texttt{null}.\mathit{created} \doteq \{\texttt{heap} := \texttt{hPre}\}\texttt{null}.\mathit{created}\end{aligned}$$

- *guard is the formula*

$$\begin{aligned}&\{w\}\big(\{\texttt{heap} := h^{base}\}(\mathit{pre} \wedge \mathit{reachableState})\\ &\qquad \wedge \{\texttt{hPre} := h^{base} \parallel \texttt{heap} := h^{new}\}(\mathit{frame} \wedge \mathit{noDeallocs})\big)\end{aligned}$$

- *equal is the formula* $\mathit{obs}(h^{new}, \texttt{o}, \texttt{p}'_1, \ldots, \texttt{p}'_n) \equiv \mathit{obs}(h^{base}, \texttt{o}, \texttt{p}'_1, \ldots, \texttt{p}'_n)$*, where* $\equiv$ *stands for* $\doteq$ *if* $\mathit{obs} \in \mathcal{F}$ *and for* $\leftrightarrow$ *if* $\mathit{obs} \in \mathcal{P}$

The useDependencyContract rule adds an assumption $\mathit{guard} \rightarrow \mathit{equal}$ to the sequent, which relates the value of *obs* in the heaps h^{base} and h^{new}. Property *noDeallocs* holds for all heap changes occurring in Java programs, where objects can be created but this process cannot be undone (we do not consider garbage collection). Property *frame* expresses that the locations in *dep* have not changed when going from h^{base} to h^{new}. If *guard* holds, then the dependency contract guarantees that *obs* has the same value for both heaps. The rule is sound if for all subtypes $B \sqsubseteq A$ of the static receiver type A the proof obligation

$\mathit{CorrectDependencyContract}(\mathit{depct}, B)$ is logically valid; this is proven in [19]. Like for method contracts, we do not allow "circular" applications of the rule.

Automatic application of this rule is not as straightforward as for useMethodContract, because the rule is nondeterministic in the choice of h^{base}, and because it can be applied repeatedly, which could lead to non-termination of automatic proof search. However, we can avoid non-termination by avoiding duplicate applications of the rule for the same pair of heap terms. To avoid a finite, but large number of "unsuccessful" applications where *guard* cannot be proven, a strategy that seems to work well in practice is to apply the rule only for choices of h^{base} for which $\mathit{obs}(h^{base}, \mathtt{o}, \mathtt{p}'_1, \ldots, \mathtt{p}'_n)$ already occurs somewhere in the sequent.

We conclude our treatment of dependency contracts by returning to the example of verifying method `m` from Sect. 2. The precondition of `m` guarantees that the invariant of `l` holds initially, i.e., that $\mathit{inv}(\mathtt{heap}, \mathtt{l})$ is true. To establish the precondition of the method call `l.get(0)` in the body of `m`, we need to establish that $\mathit{inv}(\mathit{store}(\mathtt{heap}, \mathtt{this}, \mathtt{x}, t), \mathtt{l})$ also holds (for some term t). Modularity deters us from using (1) to deduce this. Instead, we apply useDependencyContract, with $\mathit{obs} = \mathit{inv}$ and $h^{base} = \mathtt{heap}$. We get the following instantiation for *guard* (already slightly simplified):

$$\begin{aligned}
&\mathit{inv}(\mathtt{heap}, \mathtt{l}) \wedge \mathit{wellFormed}(\mathtt{heap}) \wedge \mathtt{l} \neq \mathtt{null} \wedge \mathtt{l}.\mathit{created} \doteq \mathit{TRUE} \\
&\wedge \forall \mathit{Object}\, o; \forall \mathit{Field}\, f; \big((o, f) \mathrel{\dot\in} \big((\mathit{allLocs} \mathbin{\dot\setminus} \mathit{locs}(\mathtt{heap}, \mathtt{l})) \mathbin{\dot\cup} \mathit{freshLocs}(\mathtt{heap})\big) \\
&\qquad \vee \mathit{select}_{Any}(\mathit{store}(\mathtt{heap}, \mathtt{this}, \mathtt{x}, t), o, f) \\
&\qquad \doteq \mathit{select}_{Any}(\mathtt{heap}, o, f)\big) \;\wedge \mathit{noDeallocs}
\end{aligned}$$

As the only location changed between the two heaps is $(\mathtt{this}, \mathtt{x})$, and as the precondition of `m` guarantees that $(\mathtt{this}, \mathtt{x}) \mathrel{\dot\notin} \mathit{locs}(\mathtt{heap}, \mathtt{l})$ holds, we can prove that the instantiation of *guard* is satisfied. This allows us to use the instantiation of *equal*, namely $\mathit{inv}(\mathit{store}(\mathtt{heap}, \mathtt{this}, \mathtt{x}, t), \mathtt{l}) \leftrightarrow \mathit{inv}(\mathtt{heap}, \mathtt{l})$, to prove that $\mathit{inv}(\mathit{store}(\mathtt{heap}, \mathtt{this}, \mathtt{x}, t), \mathtt{l})$ holds. After an analogous derivation about the dependencies of `size`, we can establish that the precondition of `get` holds, and then conclude with the help of useMethodContract that the postcondition of `m` holds.

5 Conclusions

We have presented an extension of Harel's dynamic logic from [5] that includes explicit representations of sets of heap locations and we have demonstrated how this logic can be used to support reasoning about dynamic frames style specifications. We have focused on the details of the logic and completely ignored issues of the specification interface and the implementation of the generation of proof obligations. Suffice it to say here that the whole approach has been implemented in a variant of the KeY system[1] and successfully tested on some simple examples. The implemented system in particular comprises an extension and modification of the Java Modeling Language, JML, for dynamic frames style specifications using model fields.

[1] available at `http://i12www.ira.uka.de/~bweiss/keyheap/`

References

1. M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology (JOT)*, 3(6):27–56, 2004.
2. B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer, 2007.
3. Y. Cheon, G. T. Leavens, M. Sitaraman, and S. H. Edwards. Model variables: cleanly supporting abstraction in design by contract. *Software—Practice and Experience*, 35(6):583–599, 2005.
4. E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.
5. D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. MIT Press, 2000.
6. C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.
7. I. T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In *FM 2006*, LNCS 4085, pages 268–283. Springer, 2006.
8. G. T. Leavens, K. R. M. Leino, and P. Müller. Specification and verification challenges for sequential object-oriented programs. *Formal Aspects of Computing*, 19(2):159–189, 2007.
9. K. R. M. Leino. *Toward Reliable Modular Programs*. PhD thesis, California Institute of Technology, 1995.
10. K. R. M. Leino. Specification and verification of object-oriented software. Lecture Notes, Marktoberdorf International Summer School, 2008.
11. K. R. M. Leino, A. Poetzsch-Heffter, and Y. Zhou. Using data groups to specify and check side effects. In *PLDI 2002*, pages 246–257. ACM Press, 2002.
12. B. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, 1994.
13. J. McCarthy. Towards a mathematical science of computation. In *Information Processing 1962*, pages 21–28, 1963.
14. B. Meyer. Applying "design by contract". *Computer*, 25(10):40–51, 1992.
15. P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular invariants for layered object structures. *Science of Computer Programming*, 62(3):253–286, 2006.
16. M. J. Parkinson and G. M. Bierman. Separation logic and abstraction. In *POPL 2005*, pages 247–258. ACM Press, 2005.
17. A. Poetzsch-Heffter. Specification and verification of object-oriented programs. Habilitationsschrift, Technische Universität München, 1997.
18. P. Rümmer. Sequential, parallel, and quantified updates of first-order structures. In *LPAR 2006*, LNCS 4246, pages 422–436. Springer, 2006.
19. P. H. Schmitt, M. Ulbrich, and B. Weiß. Dynamic frames in Java dynamic logic: Formalisation and proofs. Technical Report 2010-11, Department of Computer Science, Karlsruhe Institute of Technology, 2010.
20. J. Smans, B. Jacobs, and F. Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In *ECOOP 2009*, LNCS 5653, pages 148–172. Springer, 2009.
21. J. Smans, B. Jacobs, F. Piessens, and W. Schulte. An automatic verifier for Java-like programs based on dynamic frames. In *FASE 2008*, LNCS 4961, pages 261–275. Springer, 2008.
22. K. Stenzel. A formally verified calculus for full Java Card. In *AMAST 2004*, volume 3116 of *LNCS*, pages 491–505. Springer, 2004.

A Dynamic Logic for Unstructured Programs with Embedded Assertions

Mattias Ulbrich

Karlsruhe Institute of Technology
Institute for Theoretical Computer Science
D-76128 Karlsruhe, Germany
ulbrich@kit.edu

Abstract. We present a program logic for an intermediate verification programming language and provide formal definitions of its syntax and semantics. The language is unstructured, indeterministic, and has embedded assertions. A set of sound rewrite rules which allow symbolic execution of programs is given. We prove the soundness of three inference rules which can be used to resolve loops using invariants.

1 Introduction

The purpose of deductive software verification is to formally prove that a piece of code in a particular programming language behaves as specified. This can be done on the level of the programming language or after a translation to an intermediate verification language. In this paper, we will consider a minimalistic, general verification language that covers the essential features of established intermediate languages and is close to Boogie [9]. We present a program logic in the style of first-order dynamic logic (DL) for it.

DL is a program logic which embeds pieces of code within formulas. In its original presentation [8] by Harel et al., a code fragment π in a structural language gives rise to a modality $[\pi]$ which can be used as prefix to a formula ϕ. The result is the formula $[\pi]\phi$ which is true if and only if ϕ holds in every state in which the execution of π terminates. The Hoare triple $\{\psi\}\pi\{\phi\}$ can hence be written as $\psi \rightarrow [\pi]\phi$ in DL. Since every intermediate step of a symbolic execution in DL is a formula itself, this type of verification allows the alternation of symbolic execution and application of deductive inference rules. Therefore, symbolically stepping through a program provides further insight into a process which usually happens hidden in the verification condition generator. This is not only helpful to find mismatches between specification and implementation, but also particularly valuable when experimenting with new modelling or translation techniques. Other approaches use wp-calculi to automatically compute weakest preconditions. In the end, automatic generation and proving of first-order verification conditions as done by these approaches is certainly preferable, but we believe that, in the present state of research, the possibility of interaction is a valuable factor.

The design of the intermediate language requires considerable adaptations of the original DL. Moreover, we give a set of sound rewrite rules which allow symbolic execution of programs, and prove the soundness of three inference rules which can be used to deal with loops using invariants.

The remainder of this section provides an overview of related work. We present the dynamic logic in Sect. 2. The rewrite rules used to symbolically execute programs in formulas are given in Sect. 3. Gentzen-style inference rules for the treatment of loops are presented and proved correct in Sect. 4. Conclusions in Sect. 5 wrap up the paper.

1.1 Related Work

While some verification tools (e.g., [1], [12]) take advantage of the greater transparency of source code verification, most employ a special-purpose intermediate language. The *Why* language [7] and the Forge Intermediate Representation (FIR) [6], for instance, are used as target languages by various tools. Also, verification using the low level virtual machine (LLVM) format is a topic of ongoing research. Boogie [5, 9] is the most popular intermediate language and is used as back-end for many research approaches in formal verification. The Boogie verification condition generator breaks up loops using invariants in a fashion similar to this work. In [11], a Hoare-style calculus for Java Bytecode is defined. It includes a loop rule which is similar to the inference rules of Sect. 4, but is more evolved due to the higher complexity of the Java bytecode. [4] describes a wp-calculus for Java bytecode. Therein, loops are resolved by a code modification rendering the control flow acyclic prior to the wp-calculation.

HOL/Boogie [3], like this work, aims for a combination of intermediate language and interactive verification. There, the generated verification conditions can be interactively proved; their generation, however, (i.e., the symbolic execution) remains inaccessible.

2 Syntax and Semantics

In this section, we present the syntax and semantics of unstructured dynamic logic (*USDL*). It is built around a minimalistic intermediate verification language which is unstructured, indeterministic and contains embedded assertions. The logic extends untyped first-order predicate logic, but the approach can easily be transferred to sorted logics, the issue of types is orthogonal to the novelties presented here. For instance, the polymorphic type system presented in [10] could be used.

Unlike in DL where a program π can be used as a prefix $[\pi]$ to a formula, in *USDL* π and a natural number n induce an *atomic program formula* $[n;\pi]$ which is not prefix to another formula but a formula on its own. The number n is an explicitly denoted program pointer referring to the currently active statement in π. The conditions that we want to check are embedded within π. This is done because it is not always the case that we only need to examine whether

properties hold *after* the execution, but often want to ensure that properties hold at certain points *during* the execution of a program. For a program containing a division expression $1/x$ embedded in some statement, for instance, we need to verify that upon reaching this statement, x is different from 0 to ensure this program's correctness and cannot simply postpone this check to the after state of the entire code.

2.1 Syntax

USDL is an extension of first order logic with two modal additional operators. Besides the atomic program formulas, we introduce the concept of updates which are explicitly denoted value assignments to record the effect of assignment statements.

Definition 1 (Signature). *A USDL-signature Σ for is a 5-tuple*

$$\Sigma = (\mathrm{Var}, \mathrm{Fct}, \mathrm{PVar}, \mathrm{Prd}, \alpha)$$

with

- Var*: the set of logical variable symbols*
- Fct*: the non-empty set of function symbols*
- $\mathrm{PVar} \subseteq \mathrm{Fct}$*: the set of program variables*
- Prd*: the set of predicate symbols*
- $\alpha : \mathrm{Fct} \cup \mathrm{Prd} \to \mathbb{N}$*: the arity mapping*
- $\alpha(pv) = 0$ *for any program variable* $pv \in \mathrm{PVar}$

The syntax of terms, formulas and programs is given by the grammar in Fig. 1. For predicate and function application expressions, we additionally insist on a correct number of argument terms. If a predicate or function symbol s has no arguments, we write s instead of $s()$. Terminal symbols are set in *italics* and terminal literals in **bold**.

Definition 2 (Terms and Formulas). *The set Term_Σ of all terms in the signature Σ is the set of expressions which can be produced from the non-terminal "Term" in Fig. 1.*

The set Form_Σ of all formulas in the signature Σ is the set of expressions which can be produced from the non-terminal "Formula" in Fig. 1.

Let us for an example consider a *USDL*-signature Σ which contains a program variable $x \in \mathrm{PVar}$, a unary predicate symbol $positive \in \mathrm{Prd}$ and a unary function symbol $succ \in \mathrm{Fct}$. The expression

$$[0; \mathsf{goto}\ 1\ 4, \mathsf{assume}\ \neg positive(x), x := succ(x), \mathsf{goto}\ 0, \\ \mathsf{assume}\ positive(x), \mathsf{assert}\ positive(x)] \quad (1)$$

is then a valid atomic program formula in Form_Σ.

Formula ::= Formula ($\wedge$ | $\vee$ | $\rightarrow$) Formula
| $\neg$ Formula
| ($\forall$ | $\exists$) *Var* **.** Formula
| *Prd* | *Prd* **(** TermList **)** (*)
| **{** Update **}** Formula
| [*NaturalNumber* **;** Program]
| [[*NaturalNumber* **;** Program]]
| **true** | **false**

Term ::= *Var*
| *Fct* | *Fct* **(** TermList **)** (*)
| **{** Update **}** Term

TermList ::= Term | TermList **,** TermList

Update ::= *PVar* **:=** Term
| Update $\|\|$ Update

Program ::= Statement | Program **,** Program

Statement ::= *PVar* **:=** Term
| **assert** Formula
| **assume** Formula
| **goto** *NaturalNumber*
| **goto** *NaturalNumber NaturalNumber*
| **havoc** *PVar*

(*) if the length of the term list coincides with the arity of the symbol

Fig. 1. Formulas, Terms and Programs

Definition 3 (Unstructured programs). *The set of all unstructured programs Π_Σ is the set of expressions that can be produced from the non-terminal "Program" in Fig. 1. Terms and formulas that are embedded in unstructured programs must not have free variables.*

*For a given program $\pi \in \Pi_\Sigma$, $len(\pi) \in \mathbb{N}$ denotes the length (i.e., the number of statements) of π. For a natural number $i \in \mathbb{N}$, the selection $\pi[i]$ refers to the i-th statement in π if $i < len(\pi)$ and refers to the statement "*assume false*" if $i \geq len(\pi)$.*

Unlike in dynamic logic for structured programs, we need to list the entire code, also after the current statement since goto statements may refer to any statement in the program, before or after the current one. Therefore, we need an explicit program counter which indicates which is the current statement.

2.2 Semantics

We start the definition of our model-theoretic semantics by repeating the definition of first order structures.

Definition 4 (Domain, Interpretation, Variable assignment). *A* domain *$\mathcal{D}$ is a non-empty set. For a given domain $\mathcal{D}$ and a signature Σ an* interpretation

I is a mapping assigning a meaning to every predicate and function symbol in Σ, such that

- $I(f) : \mathcal{D}^{\alpha(f)} \to \mathcal{D}$ *for any* $f \in \mathrm{Fct}$
- $I(p) \subseteq \mathcal{D}^{\alpha(p)}$ *for any* $f \in \mathrm{Prd}$

A variable assignment $\beta : Var \to \mathcal{D}$ *is a mapping from the logical variables to elements in the domain.*

The set of all interpretation functions for a given $\mathcal{D}$ and Σ is denoted by $\mathcal{I}_{\Sigma,\mathcal{D}}$.

For the notion of the state of an execution of an unstructured program, we need a way to refer to the current position within the sequence of statements, i.e. a program counter pointing to the active statement.

Definition 5 (State). *For a signature Σ and a domain $\mathcal{D}$, the set of of states $\mathcal{S}_{\Sigma,\mathcal{D}} := \mathcal{I}_{\Sigma,\mathcal{D}} \times \mathbb{N}$ is the Cartesian product of interpretations (current variable state) and natural numbers (current position in the program).*

We explicitly encode the current statement number within the execution state as it simplifies the definition of state transitions considerably if the execution environment includes a reference to the statement to be executed next.

Definition 6 (Function overriding). *Given a function $f : A \to B$ and values $a \in A$ and $b \in B$ the* function overriding $f_a^b : A \to B$ *is the function with*

$$f_a^b(x) = \begin{cases} b & \text{if } x = a \\ f(x) & \text{otherwise} \end{cases} .$$

An update $c_1 := t_1 \| \ldots \| c_n := t_n$ can be applied to an interpretation I resulting in the multiply overridden interpretation

$$I^{c_1:=t_1\|\ldots\|c_n:=t_n} := ((I_{c_1}^{val_{I,\beta}(t_1)}) \ldots)_{c_n}^{val_{I,\beta}(t_n)}$$

in which the program variables $c_1, \ldots, c_n$ have their values updated.

Definition 7 (Term evaluation). *For a given signature Σ, a domain $\mathcal{D}$, an interpretation I and a variable assigment β, the term valuation* $\mathrm{val}_{I,\beta} : Term_\Sigma \to \mathcal{D}$ *is defined by:*

- $\mathrm{val}_{I,\beta}(x) = \beta(x)$ *if* $x \in \mathrm{Var}$,
- $\mathrm{val}_{I,\beta}(f(t_1, \ldots, t_k)) = I(f)(\mathrm{val}_{I,\beta}(t_1), \ldots, \mathrm{val}_{I,\beta}(t_k))$ *if* $f \in \mathrm{Fct}$ *with* $\alpha(f) = k$ *and* $t_1, \ldots, t_k \in Term_\Sigma$,
- $\mathrm{val}_{I,\beta}(\{\mathcal{U}\}t) = val_{I^\mathcal{U},\beta}(t)$

For the definition of the semantics of atomic program formulas, the semantics of programs has to be defined. The next two definitions for programs and formulas (Def. 8 and 9) depend on each other and have to be read as one.

Definition 8 (Program execution, Traces). *The program execution function $R_\pi : \mathcal{S}_{\Sigma,\mathcal{D}} \to \mathbb{P}(\mathcal{S}_{\Sigma,\mathcal{D}})$ is a mapping that for a program $\pi \in \Pi_\Sigma$ assigns to every state a set of successor states. Its result depends on the currently active statement.*

Let $s = (I, n) \in \mathcal{S}_{\Sigma,\mathcal{D}}$ be a state and β a variable assignment. Then the value of $R_\pi(s)$ is according to the following table:

If $\pi[n]$ ***matches***	***and***	***then*** $R_\pi(s) =$
skip		$\{(I, n+1)\}$
$c := t$		$\{(I_c^{\mathrm{val}_{I,\beta}(t)}, n+1)\}$
assert ϕ	$I, \beta \models \phi$	$\{(I, n+1)\}$
assert ϕ	$I, \beta \not\models \phi$	$\emptyset$
assume ϕ	$I, \beta \models \phi$	$\{(I, n+1)\}$
assume ϕ	$I, \beta \not\models \phi$	$\emptyset$
goto m		$\{(I, m)\}$
goto m k		$\{(I, m), (I, k)\}$
havoc c		$\{d \in \mathcal{D} \bullet (I_c^d, n+1)\}$

- *We call a sequence $(s_0, s_1, \ldots, s_r)$ (or $(s_0, s_1, \ldots)$ resp.) with $s_i \in \mathcal{S}$ and $s_{i+1} \in R_\pi(s_i)$ for $i \in \{0, ..., r-1\}$ (resp. $i \in \mathbb{N}$) a finite (infinite)* trace *of π starting in s_0.*
- *We call a finite trace* maximal *if $R_\pi(s_r) = \emptyset$.*
- *A maximal finite trace $(s_0, s_1, \ldots, s_r)$ with $s_r = (I_r, n_r)$ is called* successful *if $\pi[n_r]$ is not an "*assert ...*" statement.*

Unstructured programs are indeterministic, hence, there may be no, one or many successor states in $R_\pi(s)$ to a state s. Two types of indeterminism can be distinguished: control indeterminism (induced by goto statements with two targets) and data indeterminism (induced by havoc statements which take many possible assignments into account). It seems counter-intuitive that the successor states of assert and assume statements are identical. The difference is that a trace is considered successful if it fails at an assumption but unsuccessful for a failed assertion.

Definition 9 (Formula evaluation). *For given Σ, I, β, π and $\mathcal{D}$, the validity of a formula $\phi \in Form_\Sigma$ under the given parameters is defined as:*

- $I, \beta \models \mathrm{true}$ *and* $I, \beta \not\models \mathrm{false}$
- $I, \beta \models \phi\,(\wedge | \vee | \to)\,\psi$ *iff* $I, \beta \models \phi$ *and/or/implies* $I, \beta \models \psi$.
- $I, \beta \models (\forall|\exists)x.\phi$ *iff* $I, \beta_x^d \models \phi$ *for every/some* $d \in \mathcal{D}$.
- $I, \beta \models p(t_1, \ldots, t_k)$ *iff* $(\mathrm{val}_{I,\beta}(t_1), \ldots, \mathrm{val}_{I,\beta}(t_k)) \in I(p)$ *for a predicate symbol* $p \in \mathrm{Prd}$ *with* $\alpha(p) = k$ *and* $t_1, \ldots, t_k \in Term_\Sigma$.
- $I, \beta \models \{\mathcal{U}\}\phi$ *iff* $I^{\mathcal{U}}, \beta \models \phi$
- $I, \beta \models [n; \pi]$ *iff every maximal finite trace* $(I, n), \ldots, (I_k, n_k)$ *is successful.*
- $I, \beta \models [[n; \pi]]$ *iff* $I, \beta \models [n; \pi]$ *and there is no infinite trace of π starting in* (I, n).

Let us revisit example (1) considering an interpretation with the domain $\mathcal{D} = \mathbb{Z}$, $I(succ)(n) = n + 1$ and $I(positive) = \mathbb{N}$. If $I(x) = -1$, we have the maximal trace $(I, 0), (I, 4)$ which is successful since the last considered statement $\pi[4]$ was not an assertion but an assumption. We are not interested in a further execution of this trace and regard it as "not relevant" since an assumption has proved to be false.

USDL possesses expressive means to model both partial and total correctness of code pieces using the operators $[\cdot]$ and $[[\cdot]]$. Please note that they are not dual to another like $\Box$ and $\Diamond$ in modal logics or $[\cdot]$ and $\langle\cdot\rangle$ in classical dynamic logic are.

The programming language of *USDL* has a number of points in common with regular programs upon which the while-language in dynamic logic has been defined in [8]. The program operators $\cup$ (nondeterministic choice) and * (nondeterministic repetition) are closely related to the indeterministic goto statement. The statement assume ϕ has the same semantics as the regular program $\phi?$. Harel et al. also propose an extension with wildcard assignments like $x :=?$ which is the same as the statement havoc x.

We can, hence, use the kinds of statement defined in this document to define compound structures as macros like Harel did using regular programs. Formula (1) could then be reformulated as

$$[0; \mathsf{while}\ \neg \mathit{positive}(x)\ \mathsf{do}\ x := \mathit{succ}(x)\ \mathsf{end}; \mathsf{assert}\ \mathit{positive}(x)] \tag{2}$$

using such a macro for the while-do-end loop.

It is obvious that any formula in dynamic logic without embedded assertions can canonically be translated into a formula with embedded conditions. We formulate a typical proof as $P \rightarrow [\pi]Q$ for a precondition P, and a postcondition Q. We would formulate the same problem in *USDL* as

$$P \rightarrow [0; (\pi, \mathsf{assert}\ Q)]$$

which checks property Q after the execution of π. We can also prepend the program with the statement assume P embedding also the precondition into the program and obtain the equivalent formula

$$[0; (\mathsf{assume}\ P, \pi, \mathsf{assert}\ Q)]\ .$$

Note that now the entire verification obligation is encoded within the program to be verified.

3 Symbolic Execution

We now present a set of rewriting rules which allow us to symbolically execute an unstructured program step by step, either interactively or in an automatic proof process. Unlike wp-calculi which traverse programs from back to front, we process programs in the order of an execution, beginning at the first statement.

The update mechanism allows us to record the state changes we collect during the execution. This forward treatment is particularly helpful if the execution is part of an interactive verification process since the verifier can then track more conveniently what has happened.

A rewrite rule $l \rightsquigarrow r$ allows the calculus to replace any occurrence of l within a formula with r to obtain an equivalent formula. Such a rule is sound if the formula $l \leftrightarrow r$ is valid. A rule schema of the form $C(X) \implies l(X) \rightsquigarrow r(X)$ with a set of schematic variables X is an abbreviation for the set $\{l(x) \rightsquigarrow r(x) \mid C(x)\}$ of all instances for which the (meta) condition C holds.

Theorem 10 (Symbolic execution). *The following rules are sound rewrite rules for the symbolic execution of unstructured programs.*

$$\pi[n] = \mathsf{skip} \implies [n;\pi] \rightsquigarrow [n+1;\pi] \tag{3}$$
$$\pi[n] = c := v \implies [n;\pi] \rightsquigarrow \{c := v\}[n+1;\pi] \tag{4}$$
$$\pi[n] = \mathsf{havoc}\ c \implies [n;\pi] \rightsquigarrow \forall x.\{c := x\}[n+1;\pi] \tag{5}$$
$$\pi[n] = \mathsf{goto}\ m \implies [n;\pi] \rightsquigarrow [m;\pi] \tag{6}$$
$$\pi[n] = \mathsf{goto}\ m\ k \implies [n;\pi] \rightsquigarrow [m,\pi] \wedge [k;\pi] \tag{7}$$
$$\pi[n] = \mathsf{assume}\ \phi \implies [n;\pi] \rightsquigarrow \phi \rightarrow [n+1;\pi] \tag{8}$$
$$\pi[n] = \mathsf{assert}\ \phi \implies [n;\pi] \rightsquigarrow \phi \wedge [n+1;\pi] \tag{9}$$

Proof. The soundness proofs for these rules are straightforward. We exemplarily provide them for (8) and (9). The basic argument is the same for all cases: We reduce the case that all finite traces starting in (I,n) must be successful to the case that all finite traces from $(I',n') \in R_\pi(I,n)$ are successful and encode the knowledge on I' either into an update, an implication or conjunction. The state successor relation R_π of assert and assume are identical, but their semantics differ due to the definition of successful traces.

assume: Assume $I,\beta \not\models \phi$, then $R_\pi(I,n) = \emptyset$ and the only trace beginning in (I,n) ends there in an assume statement and, hence, *is* successful. If, on the other hand, $I,\beta \models \phi$, the truth value depends entirely on the traces starting in $(I,n+1)$, therefore, on $[n+1;\pi]$.

$$\begin{aligned}
& I,\beta \models [n;\pi] \\
\iff & \text{every finite trace beginning in } (I,n) \text{ is successful} \\
\iff & I,\beta \not\models \phi \text{ or} \\
& I,\beta \models \phi \text{ and every finite trace beginning in } (I,n+1) \text{ is successful} \\
\iff & I,\beta \not\models \phi \text{ or every finite trace beginning in } (I,n+1) \text{ is successful} \\
\iff & I,\beta \models \phi \rightarrow [n+1;\pi]
\end{aligned}$$

assert: If $I,\beta \not\models \phi$, the only trace beginning in (I,n) ends there in an assert statement and, hence, is not successful. The other case depends again on the

runs from $(I, n+1)$:

$$
\begin{aligned}
& I, \beta \models [n; \pi] \\
\iff & \text{every finite trace beginning in } (I, n) \text{ is successful} \\
\iff & I, \beta \models \phi \text{ and every finite trace beginning in } (I, n+1) \text{ is successful} \\
\iff & I, \beta \models \phi \wedge [n+1; \pi]
\end{aligned}
$$

□

The presented rules execute one single step and reduce a formula to a formula encoding *all* possible follow-up traces. This implies that the traces of the atomic program formulas on the left-hand-side are finite if and only if all traces of all modalities on the right-hand-side are finite. This observation leads to

Corollary 11. *We obtain sound rewrite rules if we replace every occurrence of a modality* $[n; \pi]$ *by the corresponding terminating counterpart* $[[n; \pi]]$ *in* (3)–(9).

4 Invariant Rules

The rewrite rules in Thm. 10 and Cor. 11 allow us to symbolically execute an unstructured program in a stepwise manner. If a program contains no loops, symbolic execution eventually results in a formula free of atomic program formulas. However, as soon as the program flow allows a statement to be executed more than once during the run of a program, these rules can no longer remove atomic program formulas entirely. A calculus for symbolic execution requires rules using loop invariants to resolve programs with loops. Such rules will, naturally, closely resemble invariant rules which are used to resolve loops in structured programs.

First, we give the simple version of an invariant rule. Then, a rule involving termination is defined and, finally, a rule which preserves more context information. The latter two could canonically be combined to a rule with termination and context preservation.

4.1 Program Modifications

In classic dynamic logic, the invariant rule introduces new proof goals on the loop body, i.e. on a program which is a strict subprogram of the original code. We are not able to reduce the code to a subset of statements in *USDL* since no restriction is imposed on the targets of goto statements and any statement, also outside the loop body, may be addressed.

We need, however, a means to reduce the number of runs of a loop body to one. This is achieved by inserting new statements into the program under inspection. The insertion is problematic, however, since index changes may make goto statements point to wrong targets afterwards. To compensate for this effect, we introduce an offset correction function off_m^k which increments the target indices by k if they lie above the insertion point m.

$$
\mathit{off}_m^k(a) = \begin{cases} a & \text{if } a \leq m \\ a + k & \text{otherwise} \end{cases}
$$

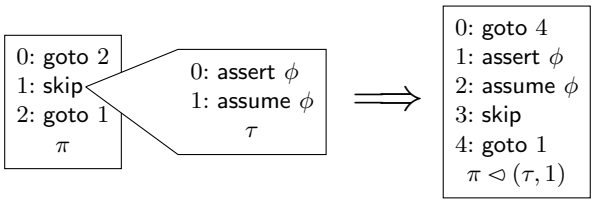


Fig. 2. Example of a program insertion

We apply off^k_m also to statements where it operates only on the target indices of goto statements and behaves like the identity function on all other statements.

Definition 12 (Statement insertion). *For programs $\pi, \tau \in \Pi$ and an arbitrary index $m \in \mathbb{N}$, the insertion $\pi \lhd (\tau, m) \in \Pi$ of τ into π at position m is defined as*

$$\big(\pi \lhd (\tau, m)\big)[i] = \begin{cases} \mathit{off}^{len(\tau)}_m(\pi[i]) & \text{for } i < m \\ \tau[i-m] & \text{for } m \le i < m + len(\tau) \\ \mathit{off}^{len(\tau)}_m(\pi[i - len(\tau)]) & \text{for } m + len(\tau) \le i \end{cases} .$$

τ is not subject to an offset correction since the programs we use for insertion in this section will not contain goto statements.

Fig. 2 shows a sample program insertion. The program $\tau = (\mathsf{assert}\ \phi; \mathsf{assume}\ \phi)$ is inserted into the program $\pi = (\mathsf{goto}\ 2; \mathsf{skip}; \mathsf{goto}\ 1)$ at position 1. Please note that in statement $4 : \mathsf{goto}\ 1$ of the resulting program, the target has *not* been incremented and still refers to the insertion point even though the statement to which it points has been changed.

Due to the index adaption off^k_m, a trace for π which does not pass through the insertion point m induces a trace for the program after insertion also (of course with possibly adapted statement indices). The only way to enter the inserted statement sequence is to reach statement m, either as a goto target or by "walking" into it. Hence, if m is not part of the trace, we can observe:

Property 13. For any trace $(I_0, k_0), \ldots, (I_r, k_r)$ with $k_i \neq n$ for $0 < i \le r$, the sequence $(I_0, k'_0), \ldots, (I_r, k'_r)$ with $k'_i = \mathit{off}^{len(\tau)}_n$ is a trace for $\pi \lhd (\tau, n)$.

The rules we develop in this section will be inference rules for a *sequent calculus*. A sequent is of the form $\Gamma \vdash \Delta$ in which the *antecedent* Γ and the *succedent* Δ are finite sets of formulas. It has the same truth value as the formula $(\bigwedge \Gamma) \to (\bigvee \Delta)$.

One problem that is not present in structured dynamic logic but with which we have to cope here is the detection of loops. In classic dynamic logic, a loop can be identified syntactically as a statement initiated with the "while" keyword. We do not have such landmarks in an unstructured program. A loop becomes a loop because of a goto statement targeting backward. Not every such statement, however, is necessarily an indicator for a loop. Therefore, we formulate our invariant rules in such a manner that they can be applied to *every* statement. Of

course, the application is not equally expedient for all execution states, and it is the task of either a static analysis or the translation mechanism to identify (and to mark) the spots at which an invariant rule should be applied.

4.2 Simple Invariant Rule

The general idea in the upcoming invariant rules is it to change the code of the program in such a way that if the starting statement n is reached again during symbolic execution, the invariant is asserted and the execution then terminated. For that purpose we insert the program (assert ϕ; assume false) into the program under inspection at the current position.

Theorem 14. *The rule*

$$\frac{\Gamma \vdash \{\mathcal{U}\}\psi, \Delta \qquad \psi \vdash [n+2; \rho_1]}{\Gamma \vdash \{\mathcal{U}\}[n; \pi], \Delta}$$

with $\rho_1 = \pi \lhd ((\mathsf{assert}\ \psi; \mathsf{assume}\ \mathrm{false}), n)$ *is a sound rule for any formula* ψ*.*

This rule has two premisses: The first provides evidence that the invariant ψ holds initially when arriving in the current state. The second premiss requires that in a state in which the invariant holds, the execution of the changed program is successful. Please note that the antecedent and succedent contexts Γ and Δ are not present in the second premiss. We will address this issue in Thm. 17.

This rule is similar to the invariant rule for a dynamic logic for a simple while language. One difference is that, here, we do not have *three* but only *two* premisses to establish. This is due to the fact that multiple assertions are embedded into the program ρ_1 and the second premiss $[n+2; \rho_1]$ plays two roles: Firstly, it proves the absence of assertion violations after the loop (the 'use case' of ψ) and, secondly, it ensures that the loop body preserves ψ establishing it as an invariant.

Proof. We can without loss of generality[1] assume that $\Delta = \emptyset$. Moreover, we may assume that (A) $\bigwedge\Gamma \to \{\mathcal{U}\}\psi$ and (B) $\psi \to [n+2; \rho_1]$ are valid formulas. For an arbitrary interpretation[2] I, we need to show that $I \models \bigwedge\Gamma \to \{\mathcal{U}\}[n; \pi]$. If $I \not\models \bigwedge\Gamma$, we are done. Thus, let $I \models \bigwedge\Gamma$. It remains to be shown that $I \models \{\mathcal{U}\}[n; \pi]$. Setting $I_{k_0} := I^{\mathcal{U}}$ yields, equivalently, $I_{k_0} \models [n; \pi]$.

Let us look at an arbitrary maximal finite trace now. We can divide this trace in "loops to n", i.e., we split the trace into r subsequences such that every occurrence of n starts a new subtrace. For any $0 \leq i < r$, the state (I_{k_i}, n) initiates a subtrace. The last trace ends in state (I_{k_r}, s_{k_r}). See Fig. 3 for an illustration.

We now claim that for every first state (I_{k_i}, n) of a subtrace, $I_{k_i} \models \psi$ holds and show this by induction on $0 \leq i < r$. For $I_{k_0} (= I^{\mathcal{U}})$, this is a simple

[1] There are first order inference rules that allow us to move the negation of all formulas in Δ to the antecedent Γ.

[2] For the sake of better readability, we leave variable assignments aside in this section.

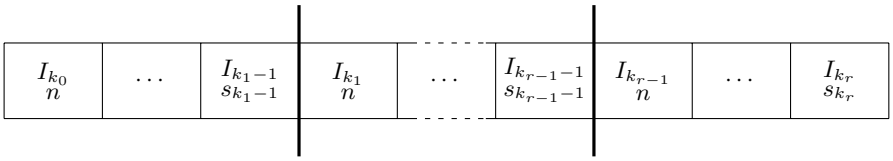


Fig. 3. Chopping a trace into subtraces

consequence of the validity of (A). Now, we assume that $I_{k_i} \models \psi$ for some $0 \leq i < r-1$.

For the trace $(I_{k_i}, n), \ldots, (I_{k_{i+1}-1}, s_{k_{i+1}-1})$, apart from the first state, no state is in statement n: it matches the requirements of Prop. 13, and, thus, we know that $(I_0, n+2), \ldots, (I_{k_{i+1}-1}, \mathit{off}^2_n(s_{k_{i+1}-1}))$ is a trace for program ρ_1. From the original trace we know that $(I_{k_{i+1}}, n)$ is a successor state to the last state of this trace. Furthermore, $\rho_1[n] = \mathsf{assert}\ \psi$ and every maximal finite trace for ρ_1 is successful by assumption (B). This implies directly that the assert-condition must be true, i.e. that $I_{k_{i+1}} \models \psi$.

We have seen now that every subtrace begins in an interpretation in which ψ holds. In particular, we have $I_{k_{r-1}} \models \psi$. The last subtrace $(I_{k_{r-1}}, n), \ldots, (I_{k_r}, s_{k_r})$ is maximal (since the entire trace was chosen maximal). Statement n does not appear after the first state of this trace. We can therefore apply Prop. 13 again and obtain a trace $(I_{k_{r-1}}, n+2), \ldots, (I_{k_r}, \mathit{off}^2_n(s_{k_r}))$ which is maximal again. Due to assumption (B), this trace must be successful, implying that the entire trace is successful. □

4.3 Invariant Rule with Termination

Thm. 14 is not sufficient if we want to incorporate the question of termination into the verification process. The rule for the terminating modality $[[\cdot]]$ introduces a variant term whose value strictly decreases from iteration to iteration. We assume there is a binary predicate symbol $\prec \in \mathrm{Prd}$ whose interpretation is a well-founded relation. With the aid of this predicate symbol, we can formulate an invariant rule which includes termination.

Theorem 15. *The rule*

$$\frac{\Gamma \vdash \{\mathcal{U}\}\psi, \Delta \qquad \psi \vdash \{nc := var\}[[n+2; \rho_2]]}{\Gamma \vdash \{\mathcal{U}\}[[n; \pi]], \Delta}$$

with $\rho_2 = \pi \lhd ((\mathsf{assert}\ \psi \wedge var \prec nc; \mathsf{assume}\ \mathrm{false}), n)$ *is a sound rule for any formula* ψ*, any term* var*, and a program variable* nc *which does not yet appear elsewhere on the sequent.*

Proof. Partial correctness $[n; \pi]$ is a direct consequence of Thm. 14 since we made the program modification *stronger* requiring $\psi \wedge var \prec nc$ to hold instead of only ψ.

Like in the proof above, we fix an interpretation I with $I \models \bigwedge \Gamma$ and set $I_{k_0} := I^{\mathcal{U}}$. It remains to be shown that there is no infinite trace for π starting in (I_{k_0}, n). Assuming there is such an infinite trace, we could subdivide it into subtraces such that every occurrence of the statement n initiates a new subtrace like in the previous proof. We can use the induction from the proof of Thm. 14 to establish that for every first state (I_{k_i}, n) of a subtrace we have $I_{k_i} \models \psi$.

In case there are finitely many subtraces, the last subtrace $\big((I_{k_{r-1}}, n), \dots\big)$ must be infinitely long and does not pass through n. We have $I_{k_{r-1}} \models \psi$ which already contradicts the second premiss which forbids an infinite trace for π starting in $(I_{k_{r-1}}, n)$ (because it uses the operator for total modality).

In case of infinitely many subtraces, every subtrace is finite. For the first states of the subtraces, we define $v_i := \mathrm{val}_{I_{k_i}}(var)$. If we take one beginning state (I_{k_i}, n) with $i > 0$, we know that (*) $I_{k_i} \models var \prec nc$ since this formula is part of the asserted loop invariant. As nc does not occur elsewhere on the sequents and because of the semantics of the update $nc := var$, we get that nc holds the value of var of the previous iteration, i.e. $I_{k_i}(nc) = v_{i-1}$. This and (*) imply that $(v_{i-1}, v_i) \in I(\prec)$. The sequence $(v_1, v_2, \dots)$ would therefore be an infinitely descending chain for $I(\prec)$ which cannot be since $\prec$ was chosen as a well-founded relation. □

4.4 Improved Invariant Rule

The major disadvantage of the rules in Thms. 14 and 15 is that the information contained in Γ and Δ of the conclusion is not available in the second premiss. There invariant ψ is the only formula in the antecedent of the sequent. If any of the information encoded in $\Gamma \cup \Delta$ was needed to close the proof, it would have to be implied by ψ and one would need to proof its validity.

We will provide an invariant rule which keeps the context Γ and Δ but subjects those program variables which are touched during a loop iteration to a generalisation. We can use the havoc statement to do this generalisation because of (5).

The rule follows the ideas of [2] were a context preserving invariant rule is defined for a structured dynamic logic. The advantage is that more information on the sequent remains available and does not need to be encoded in the invariant.

Definition 16 (loop-reachable). *A statement m is called* loop-reachable *from n within a program π if there is a trace $(I_o, k_0), (I_1, k_1), \dots$ such that*

1. *$k_o = n$,*
2. *there is an index $r \geq 1$ with $k_r = m$, and*
3. *there is an index $s > r$ with $k_s = n$.*

We denote this as $reach(n, m, \pi)$.

We use the notion of reachability to define the set of possibly modified program variables as

$$mod(n, \pi) := \left\{ c \;\middle|\; \begin{array}{l} \text{there are } m, c \text{ and } t \text{ s.t. } reach(n, m, \pi) \text{ and} \\ (\pi[m] = \mathsf{havoc}\ c \text{ or } \pi[m] = c := t) \end{array} \right\} \subseteq \mathrm{PVar} \quad .$$

Loop reachability can, in general, not be computed. The reachability of a statement may depend on the satisfiability of an assumption statement earlier in the execution path and this is undecidable. However, a static analysis can be used to over-approximate $mod(n, \pi)$.

The modified program ρ_3 is now more complex. The first two statements have the same intention as in Thm. 14 and the concluding assumption corresponds to the formula ψ in the antecedent of the second premiss in rule Thm. 14. The remaining statements need to be added to anonymise the values of those program variables that are possibly changed by the execution of the loop body.

Theorem 17. *The rule*

$$\frac{\Gamma \vdash \{\mathcal{U}\}\psi, \Delta \qquad \Gamma \vdash [n+2; \rho_3], \Delta}{\Gamma \vdash \{\mathcal{U}\}[n; \pi], \Delta}$$

with

$$\rho_3 \;=\; \pi \lhd ((\mathsf{assert}\ \ \psi; \mathsf{assume}\ \ \text{false}; \mathsf{havoc}\ \ r_1; \ldots; \mathsf{havoc}\ \ r_b; \mathsf{assume}\ \ \psi), n)$$

is a sound rule for any formula ψ and any finite set $\{r_1, \ldots, r_b\}$ with $mod(n; \pi) \subseteq \{r_1, \ldots, r_b\} \subseteq \mathrm{PVar}$.

Proof. Again, let $\Delta = \emptyset$. We observe that the second premiss is (after a number of steps of symbolic execution and simplification) equivalent to

$$\Gamma \vdash \forall x_1. \ldots \forall x_b. \{r_1 := x_1 \| \ldots \| r_b := x_b\}(\psi \to [n + 2 + b + 1; \rho_3])$$

which by construction (the inserted havoc and following assume statements cannot be executed again) is equivalent to

$$\Gamma \vdash \forall x_1. \ldots \forall x_b. \{r_1 := x_1 \| \ldots \| r_b := x_b\}(\psi \to [n + 2; \rho_1])$$

For an interpretation I with $I \models \bigwedge \Gamma$, we know, because of the validity of the premiss, that

$$I \models \forall x_1. \ldots \forall x_b. \{r_1 := x_1 \| \ldots \| r_b := x_b\}(\psi \to [n + 2; \rho_1]) \ .$$

If an interpretation I' differs from I at most on the values of the program variables $r_1, \ldots, r_b$, then we have due to the semantics of the quantifier and the updates that also

$$I' \models (\psi \to [n + 2; \rho_1]) \ .$$

For a trace for $[n; \pi]$ (cf. Fig. 3) we observe that every statement before $(I_{k_{r-1}}, n)$ is loop-reachable from n. The program variables which are changed over this trace are, hence, in $mod(n, \pi)$ and, therefore, also among the $\{r_1, \ldots, r_b\}$. This implies that for all $0 \leq i < r$, the interpretation I_{k_i} coincides with I on the required program variables and we obtain $I_{k_i} \models (\psi \to [n + 2; \rho_1])$ and, hence, $I_{k_i} \models [n + 2; \rho_1]$ by induction from the proof of Thm. 14.

In particular we have $I_{k_{r-1}} \models [n + 2; \rho_1]$ for which we saw in the proof of Thm. 14 that it implies that the entire trace is successful. □

5 Conclusion

In this paper, we have presented a dynamic logic *USDL* for an unstructured verification language. The logic differs from Harel's logic as presented in [8] as it contains the formulas to be verified embedded in the program code. We have provided a model-theoretic semantics for *USDL* and calculus rules for the symbolic execution of programs within *USDL* formulas. For the treatment of loops, we have proved the soundness of three invariant rules.

Future work on this topic includes the examination of the relationship between a propositional variant of the logic and propositional dynamic logic (PDL).

The presented calculus has been implemented in an interactive, rule-based proof-of-concept tool which has been used to successfully conduct first experiments on the benefits of interaction in verification with intermediate languages.

Acknowledgements The author would like to thank Peter H. Schmitt for his constructive comments which helped improve this paper.

References

1. B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer-Verlag, 2007.
2. B. Beckert, S. Schlager, and P. H. Schmitt. An improved rule for while loops in deductive program verification. In *Seventh Intl. Conf. on Formal Engineering Methods (ICFEM)*, pages 315–329. Springer-Verlag, 2005.
3. S. Böhme, K. R. M. Leino, and B. Wolff. HOL-Boogie—An interactive prover for the Boogie program-verifier.
4. L. Burdy and M. Pavlova. Java bytecode specification and verification. Manuscript, 2005.
5. R. DeLine and K. R. M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70., Microsoft Research, 2005.
6. G. Dennis, F. S.-H. Chang, and D. Jackson. Modular verification of code with SAT. In *ISSTA '06: Proceedings of the 2006 international symposium on Software testing and analysis*, pages 109–120, New York, NY, USA, 2006. ACM.
7. J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *In CAV '07*, pages 173–177, 2007.
8. D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. MIT Press, 2000.
9. K. R. M. Leino. This is Boogie 2, 2008.
10. K. R. M. Leino and P. Rümmer. A polymorphic intermediate verification language: Design and logical encoding. In *TACAS*, pages 312–327, 2010.
11. C. L. Quigley. A programming logic for java bytecode programs. In *TPHOLs*, pages 41–54, 2003.
12. K. Stenzel. *Verification of Java Card Programs*. PhD thesis, University of Augsburg, 2005.

A Refinement Methodology for Object-Oriented Programs *

Asma Tafat[1], Sylvain Boulmé[2], and Claude Marché[3,1]

[1] Lab. de Recherche en Informatique, Univ Paris-Sud, CNRS, Orsay, F-91405
[2] Institut Polytechnique de Grenoble, VERIMAG, Gières, F-38610
[3] INRIA Saclay - Île-de-France, F-91893

Abstract. Refinement is a well-known approach for developing correct-by-construction software. It has been very successful for producing high quality code e.g., as implemented in the B tool. Yet, such refinement techniques are restricted in the sense that they forbid aliasing (and more generally sharing of data-structures), which often happens in usual programming languages.
We propose a sound approach for refinement in presence of aliases. Suitable abstractions of programs are defined by algebraic data types and the so-called model fields. These are related to concrete program data using coupling invariants. The soundness of the approach relies on methodologies for (1) controlling aliases and (2) checking side-effects, both in a modular way.

1 Introduction

Design-by-contract is a methodology for specifying programs (in particular object-oriented ones) by attaching pre- and post-conditions to functions, methods and such. In recent years, significant progress has been made in the field of deductive verification of programs, which aims at building mathematical proofs that such a program satisfies its contracts. Some widely used programming languages, like JAVA, C# or C have been equipped with formal specification languages and tools for deductive verification, e.g., JML [11] for Java, Spec# [6] for C#, ACSL [7] for C. The assertions written in the contracts are close to the syntax of the underlying programming language, and directly express properties of the variables of the program. However, for codes of large size the need for data abstractions arises, both for writing advanced specifications and for hiding implementation details.

Leavens et al. [18] have listed some specification and verification challenges for sequential object-oriented programs that still have to be addressed. One of these issues deals with data abstraction in specification, and more specifically the specification of modeling types. The task to be done is summed up as follows: *Develop a technique for formally specifying modeling types in a way that is useful for verification.*

* This work is partly supported by INRIA Collaborative Research Action (ARC) "CeProMi", `http://www.lri.fr/cepromi/`

This paper proposes to solve this problem using a refinement approach. Our proposal has strong connections with the notion of program refinement of the B method [1] for developing correct-by-construction programs. In a first step, abstract views of objects are specified with so-called *model fields* as an abstract representation of their state. Unlike the standard model fields of JML, our model fields are described as *algebraic data types* instead of immutable objects. The refinement of such an abstract view is a concrete object together with a coupling invariant that connects its concrete fields with model fields of the abstract view. Like all refinement approaches, we want to ensure that reasoning on the abstract view in a client code does not allow establishing properties that are falsified at runtime. Hence, in the presence of arbitrary pointers or references (and thus data sharing), the verification of these coupling invariants requires a strict policy on assignment, for controlling where a given invariant is potentially broken.

This paper is based on the *ownership* policy of Boogie methodology [4]. In Section 3 we propose a variant of ownership to support model fields. The main result (Theorem 1) states that class invariants, including coupling invariants, are preserved during execution. Section 4 then proposes a refinement approach for object-oriented programs, where subclasses are refined programs for abstract classes. An additional ingredient needed is a technique for controlling side effects in subclasses: in this paper we use *datagroups* [22]. We illustrate the methodology on two examples: first, the *calculator* example of Morgan [23], and second, an instance of the observer pattern.

2 Preliminaries

2.1 Deductive verification of contracts

We consider object-oriented programs equipped with a *Behavioral Interface Specification Language* (BISL) such as JML [11] for Java, Spec# [6] for C#, etc. Methods are equipped with *contracts*: pre- and postconditions, frame clauses to specify write effects, etc; and objects are equipped with *class invariants*.

Our goal is to verify that a program satisfies its specification using proof methods. A general approach for that purpose is the generation of *verification conditions* (VCs), which are logical formulas whose validity implies the correctness of the program with respect to the specification. To automatize this process, a popular method is the calculus of weakest preconditions, as available in ESC/Java [14], Spec# [6], and the Why platform [17]. In a slightly different context but for similar purposes, weakest preconditions are used in the B method [1] for developing correct-by-construction programs.

The primary application of BISL is runtime assertion checking. For this reason, assertions used in annotations are boolean expressions. However, it has been noted by several authors [12, 16] that for deductive verification purposes, the language of assertions should be instead based on classical first-order logic. In particular, it allows calling SMT provers to discharge VCs. This is the setting we assume in this paper. More generally, we assume that the specification

language allows user-defined algebraic datatypes, such as in B [1], ACSL [7] or Why [17].

Example 1. Multisets, or *bags*, are typically a useful algebraic datatype for specifying programs, that we need later. Here is a (partial) user-defined axiomatization of bags (See [26] for a full one)

```
type bag<X>;
constant emptybag: bag<X>;
function singleton: X -> bag<X>;
function union: bag<X>, bag<X> -> bag<X>;
function card: bag<X> -> integer;
function sumbag: bag<real> -> real;
axiom union_empty: \forall b:bag<X>, union(b,emptybag) = b;
axiom union_assoc: \forall b1,b2,b3:bag<X>,
       union(b1,union(b2,b3)) = union(union(b1,b2),b3);
...
```

2.2 Refinement

Refinement calculus [23, 2] is a program logic which promotes an incremental approach to the formal development of programs: from very abstract specifications down to implementations. The B method [1] has successfully mechanized this logic in some industrial developments [8]. In the B method, an abstract component introduces abstract variables and defines each procedure by an abstract behavior on these variables. A refined component is then given using other variables, a *coupling invariant* which relates them to abstract variables, and refined definitions of procedures. A component may be refined several times in this way, until all behaviors of procedures are given as programs.

Example 2. Morgan's calculator [23] is a typical and simple example of refinement. Such a calculator is aimed at recording a sequence of real numbers, and providing their arithmetic mean on demand. Below, on the left, is an abstract view of a calculator, whereas the right part presents a refinement expressing that two numbers are sufficient to encode the required informations on the whole sequence:

var $values$: $\texttt{bag}(\mathbb{R})$
init $values \leftarrow \emptyset$;
op add($x : \mathbb{R}$):void =
  $values \leftarrow values \cup \{x\}$;
op mean():$\mathbb{R}$ =
  pre $values \neq \emptyset$;
  result $\leftarrow \dfrac{\texttt{sumbag}(values)}{\texttt{card}(values)}$;

var $count$: $\mathbb{N}$
var sum : $\mathbb{R}$
invariant $sum = \texttt{sumbag}(values) \wedge$
  $count = \texttt{card}(values)$;
init $sum \leftarrow 0$; $count \leftarrow 0$;
op add($x : \mathbb{R}$):void =
  $sum \leftarrow sum + x$;
  $count \leftarrow count + 1$;
op mean():$\mathbb{R}$ =
result $\leftarrow sum/count$;

This paper investigates how to adapt this approach to reasoning on object-oriented programs. However, we consider the simpler case with only one abstract level, where behaviors are given as pre/post-conditions together with frame clauses, and one concrete level, the implementations in the underlying programming language.

Technically, refinement corresponds to the condition below, verified for each operator, where x are the input parameters, a the abstract variables, c the concrete ones, P the abstract precondition, I the coupling invariant, Q the abstract postcondition, S the body of the concrete operation: $\forall c, x, a;\ (P \wedge I) \Rightarrow \exists a'; \mathbf{wp}(S, (Q \wedge I)[a \mapsto a'])$. Let us explain this VC from client's point of view. For any reachable state c, a satisfying I in the execution of a given client code, there exists abstract values a' such that I is still satisfied. For instance, in a client code, we can safely replace an execution of the concrete sequence S, by a non-deterministic update of variable a that chooses an arbitrary value a' satisfying both Q and I. The VC on any operation call ensures that the remaining of the client code is correct for all possible choices of this non-deterministic update.

Example 3 (Calculator continued). The VC for the add operation is

$$\begin{array}{l}\forall count, sum, values, x;\ (sum = \texttt{sumbag}(values) \wedge count = \texttt{card}(values)) \Rightarrow \\ \quad \exists values'; values' = values \cup \{x\} \wedge \\ \quad\quad (sum + x = \texttt{sumbag}(values') \wedge count + 1 = \texttt{card}(values'))\end{array}$$

which is a logical consequence of the axiomatization of bags (Example 1).

2.3 Model fields

Model fields have been introduced by Leino [19] as abstract representations of object states. Syntactically, a *model field* is used only for specification purpose and remains invisible from the actual code. Clients can refer to its successive values in their assertions, without knowing how this abstract state is implemented.

We adopt the JML syntax for model fields [13], but the JML *represents* clauses are replaced by coupling invariants, which are more general since they do not enforce a model field to be deterministically determined from concrete fields. Notice that model fields differ from *ghost* fields: the latter can be directly assigned in implementations.

Example 4. In the following, we declare a public view of class `Euros` to compute addition and subtraction on euros. In this public view, the model field `value` represents the state of the object as a *real* number.

```
class Euros {
  //@ model real value=0.0;
  //@ invariant this.value>=0.0;

  /*@ assigns this.value;
    @ ensures this.value==\old(this.value+a.value);  */
  void add(Euros a);
}
```

In the corresponding implementation below, the real number is coded as two integers: in particular, the fractional part of the real is coded as a byte less than 100.

```
class Euros {
  private int euros=0;
  private byte cents=0;
  //@ invariant 0 <= euros && 0 <= cents < 100;
  //@ invariant coupling: value == euros + cents / 100.0;

  void add(Euros a) {
      euros += a.euros; cents += a.cents;
      if (cents >= 100) { euros++; cents −= 100; }
  }
}
```

Giving a semantics to model fields leads to several issues [10, 13, 20] that we will discuss further in Section 5: as model fields are not directly assigned in the code, at which program points the values of model fields are changed? At which program points the coupling invariant, relating the concrete fields (like `euros` and `cents` above) to the model field (`value` above), is ensured? Also, the public view above says that only model field `value` is modified, is it sound to ignore the change on private fields (like `euros` and `cents`) in clients?

2.4 Ownership

Checking preservation of class invariants is known to be a difficult problem because of aliasing and thus sharing of references [18]. The *ownernhip* approach proposed by Barnett et. al in 2004 [4] is suitable for deductive verification, and implemented in the Boogie VC generator [5]. Informally, *ownership* views objects as boxes which can be opened or closed. A closed object ensures that its invariant is satisfied. Conversely, the contents of an object can be updated only when this object is open. The status, open or closed, of an object is represented by some specific boolean field `inv` similar to a model field (that is only accessible in specifications). Concretely, opening and closing an object is performed by using special statements `unpack` and `pack`. Hence, closing an object generates a VC that the invariant of this object holds.

Updating an object's field must not break the invariant of an other closed object. This crucial property is ensured by a strict discipline. First, the invariant of an object o can constrain only objects accessible via dedicated fields called "`rep` fields". More precisely, the invariant of o may refer to $o.f_1 \dots f_n.g$ only if $f_1, \dots, f_n$ are declared as `rep`. Hence, a `rep` field f declares that whenever o is closed, then $o.f$ must also be closed: in this case, we say that o *owns* $o.f$. Moreover, a given closed object can only have *at most one owner*. Technically, another model boolean field `committed` represents whether an object has a owner or not. This field acts as a lock that is only modified by applying `unpack` and `pack` statements to its owner. This ensures that an object can not be modified without opening its owner first.

With inheritance, this approach is generalized by transforming `inv` field into a class name: "$o.inv = C$" means that object o satisfies invariant of all superclasses of C (C included). Packing and unpack are made relative to a class name: "`pack` o `as` C" means "close the box o with respect to class C"; whereas "`unpack` o `from` C" means "open the box o out of C", i.e set its `inv` to the superclass of C.

This informal description is formalized in next section (see also [26]), together with our proposed extension adding a specific support of model fields.

3 Ownership and Model Fields

3.1 Language setting

We consider a core object-oriented language [4] extended with model fields. A hierarchy of classes is defined together with specifications. First there is a base class `Object` which contains only the two special model fields: *inv* denoting a class name and *committed* denoting a boolean. Each class is given by:

- its (unique) name
- the name of its superclass, `Object` by default
- a set of model fields, whose types are logic datatypes
- a set of concrete fields, some of them might be marked as `rep`
- an invariant, that is a logical assertion syntactically limited to mention well-typed locations (according to Java static typing) of the form "this.$f_1 \dots f_n.g$" where f_i are `rep` concrete fields and g is either a model or a concrete field.
- a set of method definitions that consists of a profile "$\tau\ m(x_1 : \tau_1, \dots, x_n : \tau_n)$", a body, and a *contract* defined as:
  - a pre-condition $Pre_m(this, x_1, \dots, x_n)$
  - a post-condition, $Post_m(this, x_1, \dots, x_n, result)$ which might refer to the pre-state using *old* and to the return value using *result*
  - a frame clause $Assigns(locs)$ specifying the side-effects: it states that any memory locations, allocated in the pre-state, that do not belongs to *locs*, is unchanged in the post-state.
- a set of constructors with a profile $C(x_1 : \tau_1, \dots, x_n : \tau_n)$, a body, and a *contract* similar to those of methods, except that precondition cannot refer to *this* and postcondition cannot not refer to result, but can refer to *this* to denote the constructed object.

Pre- and postconditions must be purely logic expressions, in particular we forbid constructor or method calls in them. A class inherits fields of its superclass, in particular it has an *inv* and a *committed* field. We denote by $<:$ reflexive-transitive closure of subclass relation. We denote by $Comp_T$ the set of rep fields declared in class T. More precisely, $Comp_T$ contains only rep fields declared in T but not the rep fields declared in a strict superclass of T. A field update $o.f := E$ where f is a concrete field declared in superclass T of o static type, has the precondition $\neg(o.inv <: T)$, meaning that $o.inv$ must be a strict superclass

of T. Field update $o.f := E$ where f is a model field is syntactically forbidden. Using `pack` (see below) is the only way to update model fields. Bodies of methods are verified in a context where $type(this)$ is the current class: inherited methods are rechecked according to the context of the subclass.

3.2 pack/unpack for model fields

We define two statements for opening and closing object. Opening an object o is done via the following statement, whose semantics is given by the contract:

unpack o **from** T :
  pre: $o \neq null \wedge o.inv = T \wedge \neg o.committed$
  assigns: $o.inv, o.f.committed \mid f \in Comp_T$
  post: $o.inv = S \wedge \bigwedge_{f \in Comp_T} o.f.committed = false$

where T is a class identifier (using $type(o)$ instead of T is forbidden, hence $Comp_T$ is statically known by VC generator), and S is the direct superclass of T.

The **pack** statement is significantly more complex than the original in Boogie's ownership, because it performs a non-deterministic update of model fields. We adopt here a syntax inspired by unbound choice operator of B:

pack o **as** T **with** $M_0 := v_0, \ldots, M_n := v_n$ **such that** P

where o is the object to close, M_i is a model field to update, v_i is a fresh variable denoting the desired new value for $o.M_i$, and P is a proposition which can mention both v_i and the current values of the model fields or the concrete fields. Syntactically, T is a class identifier and M_i must belong to model fields declared in T (updating model fields of a superclass is forbidden). The semantics is given by the contract:

pack o **as** T **with** $M_0 := v_0, \ldots M_n := v_n$ **such that** P :
  pre: $o \neq null \wedge o.inv = S \wedge$
    $\exists v_0, \ldots, v_n,\ Inv_T[this.M_i \mapsto v_i][this \mapsto o] \wedge P \wedge$
    $\bigwedge_{f \in Comp_T} o.f = null \vee (o.f.inv = type(o.f) \wedge \neg o.f.committed)$
  assigns: $o.M_0, \ldots, o.M_n, o.inv, o.f.committed \mid f \in Comp_T$
  post: $o.inv = T \wedge Inv_T[this \mapsto o] \wedge (\mathbf{old}(P))[v_i \mapsto o.M_i] \wedge$
    $\bigwedge_{f \in Comp_T} o.f \neq null \Rightarrow o.f.committed$

where S is the superclass of T, $type(e)$ denotes the dynamic type of expression e and $Inv_T[this.M_i \mapsto v_i][this \mapsto o]$ is the coupling invariant in which model fields M_i mentioned in the clause **with** are substituted by v_i.

Example 5. Figure 1 is a variant of Morgan's calculator equipped with pack/unpack statements and pre- and postconditions to state the values of `inv` and `committed` fields. The VC generated from the precondition of pack statement in method `add` is:

$$\begin{array}{l} this \neq null \wedge this.inv = Object \wedge \\ \exists v,\ this.sum = sumbag(v) \wedge this.count = card(v) \wedge \\ \qquad v = union(this.values, singleton(x)) \end{array}$$

```
class SimpleCalc {
  //@ model bag<real> values;
  private int count;
  private double sum;
  //@ invariant sum==sumbag(values) && count==card(values);

  /*@ assigns \nothing;
    @ ensures inv==\type(this) && !committed
    @         && values == empty_bag; */
  SimpleCalc() {
      sum = 0.0; count = 0;
      /*@ pack this \as SimpleCalc \with values:=v
        @    \such_that v==empty_bag; */
  }

  /*@ requires inv==\type(this) && !committed;
    @ assigns values, count, sum;
    @ ensures values==union(\old(values),singleton(x)); */
  void add(double x) {
      //@ unpack this \from SimpleCalc;
      sum += x; count++;
      /*@ pack this \as SimpleCalc \with values := v
        @   \such_that v == union(values,singleton(x)); */
  }

  /*@ requires inv==\type(this) && values != empty_bag;
    @ assigns \nothing;
    @ ensures \result==sum_bag(values)/card(values); */
  double mean() { return sum/count; }
}
```

Fig. 1. Morgan's calculator with pack/unpack

Hence, notice that the weakest precondition of `add` is thus very similar formula to the VC of the refinement given in Example 3.

3.3 Invariant preservation

We state below our main result. The first proposition means that committed objects must be fully packed. The second states the most important property: invariants are valid for packed objects. The third states that components of a closed object are committed. The fourth expresses that a committed component can have only one owner.

```
abstract class Calc {
  //@ datagroup Gvalues;
  //@ model bag<real> values \in Gvalues;

  /*@ requires this.inv == \type(this) && !this.committed;
    @ assigns Gvalues;
    @ ensures values == union(\old(this.values),singleton(x));
    @*/
  abstract void add(double x);

  /*@ requires inv == \type(this) && values != empty_bag;
    @ assigns \nothing;
    @ ensures \result == sum_bag(values)/card(values); */
  abstract double mean();
}
```

Fig. 2. Morgan's Calculator, abstract class

Theorem 1 (invariant preservation). *The following properties hold during any program execution.*

$$\forall o; o.committed \Rightarrow o.inv = \textbf{\textit{type}}(o) \quad (1)$$
$$\forall o, T; o.inv <: T \Rightarrow Inv_T(o) \quad (2)$$
$$\forall o, T; o.inv <: T \Rightarrow \bigwedge_{f \in Comp_T} o.f = null \vee o.f.committed \quad (3)$$
$$\forall o, T, o', T'; \bigwedge_{f \in Comp_T, f' \in Comp_{T'}} (o.inv <: T \wedge o'.inv <: T' \wedge o.f \neq null \wedge o.f = o'.f') \Rightarrow (o = o' \wedge T = T') \quad (4)$$

where quantifications over references range over allocated objects.

See [26] for the proof. It is similar to the one of [4]. Differences come from the presence of model fields, coupling invariants and our extended pack statement.

4 A refinement methodology

We have a notion of model fields with a proper nondeterministic semantics, similar to abstract variables as they are used in the B method. To go further, we now describe a methodology for the development of OO programs which mimics the refinement approach. This methodology is simply a combination of our notion of model fields with datagroups as proposed by [19, 22]. We introduce this methodology below on Morgan's Calculator before considering a more complex example.

4.1 Hiding effects using datagroups in assigns clauses

Let us consider Morgan's Calculator of Example 2. We would like to mimic this example in Java by splitting class `SimpleCalc` of Fig. 1 into two classes: first,

```
class SmartCalc extends Calc {
  private int count;  //@ \in Gvalues;
  private double sum; //@ \in Gvalues;
  /*@ invariant this.sum == sumbag(this.values)
    @  && this.count == card(this.values); */

  /*@ assigns \nothing;
    @ ensures this.values == empty_bag;
    @ ensures this.inv == \type(this) && !this.committed; */
  SmartCalc() {
    sum = 0.0; count = 0;
    /*@ pack this \as Calc \with values:=c
      @    \such_that c == empty_bag;
      @ pack this \as SmartCalc;     */
  }

  void add(double x) {
    //@ unpack this \from SmartCalc;
    //@ unpack this \from Calc;
    sum += x; count++;
    /*@ pack this \as Calc \with values:=c
      @    \such_that c == union(values,singleton(x));
      @ pack this \as SmartCalc;     */
  }

  double mean() { return sum/count; }
}
```

Fig. 3. Morgan's Calculator, implementation class

an abstract class `Calc` (Fig. 2) mentioning only the model field and contracts for methods; second, an implementation `SmartCalc` (Fig. 3) using concrete fields `count` and `sum`. Two successive `unpack` or `pack` statements are needed to open or close an object from class `SmartCalc` to `Calc` then to `Object`. A key issue arises here, about the specification of side effects: the abstract class is not supposed to mention `count` and `sum` in `assigns` clauses, since those fields are not even known.

In the B method [1], a simple encapsulation mechanism of private fields ensures that their modifications can not be observed from clients. Hence, in B, it is safe to simply ignore modifications on private fields in clients, since clients cannot access them. Unfortunately, such a simple approach is not sound for OO programs. Indeed, a given object can be indirectly a client of itself via a reentrant call, and observes modifications made by this reentrant call on its own private fields. Actually such a problem would also occur in B, if mutual recursion between components was allowed.

In presence of reentrancy, we can not ignore modifications on private fields. Alternatively, [19, 22] proposes to *abstract* such modifications using *datagroups*. We use this approach in this paper since it smoothly integrates into any VC generator using classical logic (see Section 5 for further discussion). Roughly, a datagroup is a name for a set of memory locations and used in `assigns` clauses to express that all its memory locations may have been modified. The main feature of datagroups is that they can be extended in subclasses with new fields (public or private). The inclusion of a field to a datagroups must appear in the declaration of that field and is defined all over its scope. Datagroups may also include other datagroups (hence, we may have nested datagroups) and a field may belong to several datagroups.

Hence, coming back to Morgan's calculator, we introduce a datagroup called `Gvalues` that consists of model field `values` in abstract class `Calc` of Fig. 2, and which is extended with concrete fields `count` and `sum` in its implementation `SmartCalc` of Fig. 3. Of course, on this example, it would be more user-friendly to identify syntactically the datagroup `Gvalues` and the model field `values`. However, in this paper, we prefer to keep a clear distinction between the two notions, since in other examples, a datagroup may contain several model fields.

4.2 Modular Reasoning on Shared State: the Observer Pattern Example

In the literature (see for instance [24]), ownership discipline is often considered as incompatible with modular reasoning on a shared state between objects. Indeed, at first sight, ownership discipline forbids objects constraining *simultaneously* a given substate through an invariant. A contribution of our work is to show that this common belief is wrong. Ownership extended with nondeterministic refinement of model fields allows some modular reasoning on a *shared state* between objects.

We illustrate this claim on *observer pattern*, a generic implementation of *event programming* in OO languages. In this pattern, an object, called Subject,

maintains a list of its dependents, called observers, and notifies them automatically of any state changes, by calling their `notify` methods. When notified, observers updates their own state according to the new state of Subject, usually by calling back some accessor of Subject. Hence, Subject is shared between observers. Moreover, observers are themselves shared between Subject and some clients of the whole pattern.

Here, we instantiate this pattern to define observers of a Morgan's calculator (example fully detailed in [26]). The key idea, that makes this example work with ownership discipline, is the following: *in observers, we clone an abstraction of their shared state using model fields* (below `size` and `mean`). Thus, these clones exist only in assertions, not at runtime:

```
abstract class CalcObs {
  SubjectCalc sub;

  //@ datagroup Gsubject;
  //@ model int size \in Gsubject;
  //@ model real mean \in Gsubject;

  /*@ requires this.inv == \type(this) && !this.committed;
    @ requires sub != null && sub.mc != null
    @          && sub.mc.inv==\type(sub.mc);
    @ assigns this.Gsubject;
    @ ensures size == card(sub.mc.values)
    @         && size*mean == sumbag(sub.mc.values);
    @*/
  abstract void notify();
}
```

A given object (here Subject) glues the actual shared state with its clones through an invariant. Here is an excerpt of its specification, where the important part is the `observers_notified` invariant:

```
class SubjectCalc {
  int obs_nb;
  rep CalcObs[] obs;
  //@ invariant obs_size: obs != null && 0<=obs_nb<obs.length;

  rep Calc mc;
  /*@ invariant observers_notified: mc != null &&
    @   \forall integer i; 0 <= i < obs_nb ==>
    @      obs[i] != null && obs[i].sub == this
    @       && obs[i].size == card(mc.values)
    @       && obs[i].size*obs[i].mean == sumbag(mc.values); */

  /*@ requires inv == \type(this) && !committed;
    @ assigns obs[0..obs_nb-1].Gsubject, mc.Gvalues ;
    @ ensures mc.values==union(\old(mc.values),singleton(x)); */
  void update(double x){
      //@ unpack this \from SubjectCalc;
```

```
      mc.add(x) ;
      for (int i = 0; i < obs_nb; i++) obs[i].notify();
      //@ pack this \as SubjectCalc ;
  }

  /*@ requires inv==\type(this) && !committed ;
    @ requires o!=null && o.inv==\type(o) && !o.committed;
    @ requires o.sub==this && obs_nb < obs.length ;
    @ assigns o.committed, o.Gsubject;
    @ assigns obs_nb, this.obs[\old(this.obs_nb)];
    @ ensures o.committed;
    @ ensures this.obs_nb==\old(this.obs_nb)+1
    @      && this.obs[\old(this.obs_nb)]==o; */
  void register(CalcObs o){
      //@ unpack this \from \type(this);
      this.obs[obs_nb++]=o;
      o.notify();
      //@ pack this \as \type(this) ;
  }
}
```

The observers can then be implemented independently by refining their own clone of the shared state: they can introduce a coupling invariant relating their own actual state to the clone. For observers, the possibility to update their model fields non-deterministically is crucial here. Indeed, observers update their clone when notified by Subject which has been modified in a undetermined way from their point of view. Here is an example of such an observer:

```
class Success extends CalcObs {
  boolean passed;
  //@ invariant coupling: passed==(size>=4 && mean>=10.0) ;

  void notify(){
      //@ unpack this \from Success ;
      //@ unpack this \from CalcObs ;
      /*@ pack this \as CalcObs \with size:=s, mean:=m
        @   \such_that s==card(sub.mc.values) &&
        @              s*m==sumbag(sub.mc.values); */
      passed = (sub.size() >= 4 && sub.mean() >= 10.0);
      /*@ pack this \as Success; */
  }
}
```

In conclusion, this cloning technique through model fields offers some freedom in the design of an architecture that is both compatible with ownership discipline and that fits the particular needs of the application. However, this example reveals the need of several improvements in our approach:

– We would like a more abstract interface for Subject. First, a more abstract representation of the set of observers is desirable. Second, it would be more convenient to include all internal state of observers in one datagroup of

Subject. However, the datagroups discipline (with the use of *pivot fields* [22, 26]) would then prevent access to observers from outside of Subject, which not desirable.
– This architecture would be more elegant if Subject was allowed to unpack observers: `notify` method of observers could hence be used to (re)pack them.[4] However, if we want to allow a given object `o` to be an unknown instance of a given class, we can not unpack `o`, because this would produce an uncontrolled side-effect on the committed field of `o` rep fields (which are not fully known).

5 Conclusions, Related Works and Perspectives

In 2003, Cheon et al. [13] propose foundations for the model fields in JML, which are presented as a way to achieve abstraction. Their main concern is the runtime assertion checker of JML, hence they naturally propose that model fields are Java objects as any other field (although immutable objects for obvious reasons), and not logical datatypes. Moreover, a model field is related to concrete fields by a *represents* clause which amounts to giving a function from concrete fields to the associated model field. Consequently, they cannot support non-deterministic updates of model fields as in Morgan's calculator: there is more than one bag having a given cardinal and a given sum of its elements.

In 2003, Breunesse and Poll [10] explore the possible use of model fields in the context of deductive verification instead. They also analyse the potential use of non-deterministic coupling relations via `\such_that` clauses. They propose four possible approaches. The first one, which indeed originates from Leino and Nelson [21], amounts to assume that the coupling invariant holds at any program point. This is impracticable and indeed unsound since it does not check for existence of a model. Two other approaches amount to systematically replace each predicate refering to a model field by a complex formula with proper quantifiers, these are impracticable too. The last approach replaces the model fields by an underspecified function which returns any possible value for it. In some sense it is similar to our **pack with** but clearly less flexible.

In 2006, Leino and Müller [20] proposed a technique to deal with model fields via ownership. This work was the main inspiration of ours: we wanted to remove a limitation of their approach which prevent them from dealing with Morgan's calculator. Precisely, the post-condition of their pack statement for the `add` method is just the coupling invariant

$$this.sum = sumbag(this.values) \wedge this.count = card(this.values)$$

from which it is not possible to prove the postcondition

$$this.values = union(old(this.values), singleton(x))$$

[4] Indeed, method `register` of Subject, that registers a new observer, could be called on a open observer before to pack it via `notify`. Thus, inside their constructor, observers would not be obliged to be pack in a dummy state before the call to `register`.

because the latter is not the only bag b which have the given sum and cardinal. In other words, Leino-Müller approach [20] can only deal with deterministic coupling invariants, which impose only one possible value for model field from the values of the concrete fields.

Our methodology for refinement has a few originalities: unlike previous approaches, it allows non-deterministic refinement, as it exists classically in refinement paradigm; it permits to safely hide the side-effects on private data from the public specification of classes, which is a very important property for modularity of reasoning on programs.

More recently, the Jahob verification system [29] also uses algebraic data types to model programs. However, again the relation from concrete data to abstract is done by logic functions, hence as previous approaches they are deterministic and not amenable to refinement in general.

On the other way around, there have been attempts to apply ownership systems to refinement-based techniques as in B. Boulmé and Potet [9] have shown that the ownership policy of Boogie is a strict generalization of the verification of invariants in B. More precisely, they have encoded the component language of B (without refinement) in a pseudo-Boogie language, and have shown that the VCs induced by this encoding imply those of B. Moreover, syntactic restrictions of B that limit data-sharing between components can be safely relaxed using a Boogie approach. However they have only considered B without refinement. By extending their encoding using a **pack with** statement, we can also derive the VCs of B for a subset of B limited at one level of refinement. However, extending this to several levels of refinements is not obvious.

Our refinement methodology combines modular techniques for (1) ensuring invariant preservation (ownership) and (2) checking side effects. Although such a combination was already said possible in the past [20], it seems strange that to the best of our knowledge, no tool currently propose both, e.g., Spec# has ownership but no datagroups, whereas ESC/Java2 has datagroups but no ownership.

Datagroups provide quite a simple technique to check side-effects, in particular because it naturally fits in a standard weakest precondition calculus in classical first-order logic. It is clearly interesting to investigate more recent approaches like *separation logic* [25], *dynamic frames*, or region-based access control [27, 28, 3].

In this paper we choose that model fields are algebraic data types because it is handy for deductive verification. However our refinement technique is certainly usable with immutable objets as models, more suitable for runtime verification; such as by approaches of Darvas [15] which map model classes to algebraic theories.

Acknowledgments We thank Marie-Laure Potet, Wendi Urribarri, Christine Paulin and others CeProMi members for their fruitful discussions on this work.

References

1. J.-R. Abrial. *The B-Book, assigning programs to meaning.* Cambridge University Press, 1996.
2. R.-J. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction.* Springer-Verlag, 1998.
3. A. Banerjee, D. A. Naumann, and S. Rosenberg. Regional logic for local reasoning about global invariants. In *European Conference on Object-Oriented Programming (ECOOP'08)*, Paphos, Cyprus, July 2008.
4. M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, June 2004.
5. M. Barnett, R. DeLine, B. Jacobs, B.-Y. E. Chang, and K. R. M. Leino. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *FMCO'05*, volume 4111 of *LNCS*, pages 364–387, 2005.
6. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# Programming System: An Overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS'04)*, volume 3362 of *LNCS*, pages 49–69. Springer, 2004.
7. P. Baudin, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto. *ACSL: ANSI/ISO C Specification Language*, 2008. `http://frama-c.cea.fr/acsl.html`.
8. P. Behm, P. Benoit, A. Faivre, and J.-M. Meynadier. Météor: A successful application of B in a large project. In *Formal Methods'99*, volume 1708 of *LNCS*, pages 348–387. Springer, Sept. 1999.
9. S. Boulmé and M.-L. Potet. Interpreting invariant composition in the B method using the Spec# ownership relation: a way to explain and relax B restrictions. In J. Julliand and O. Kouchnarenko, editors, *B 2007*, volume 4355 of *LNCS*. Springer, 2007.
10. C.-B. Breunesse and E. Poll. Verifying JML specifications with model fields. In *FTfJP'03*, 2003.
11. L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. 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*, 2004.
12. J. Charles. Adding native specifications to JML. In *FTfJP'06*, 2006.
13. Y. Cheon, G. Leavens, M. Sitaraman, and S. Edwards. Model variables: cleanly supporting abstraction in design by contract. *Softw. Pract. Exper.*, 35(6):583–599, 2005.
14. D. R. Cok and J. Kiniry. ESC/Java2: Uniting ESC/Java and JML. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *CASSIS*, volume 3362 of *LNCS*, pages 108–128. Springer, 2004.
15. A. P. Darvas. *Reasoning About Data Abstraction in Contract Languages.* PhD thesis, ETH Zurich, 2009.
16. J.-C. Filliâtre and C. Marché. Multi-prover verification of C programs. In *ICFEM'04*, volume 3308 of *LNCS*, pages 15–29. Springer, 2004.
17. J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *CAV'07*, volume 4590 of *LNCS*, pages 173–177, Berlin, Germany, July 2007. Springer.
18. G. T. Leavens, K. R. M. Leino, and P. Müller. Specification and verification challenges for sequential object-oriented programs. *Formal Aspects of Computing*, 2007.

19. K. R. M. Leino. Data groups: Specifying the modification of extended state. In *OOPSLA'98*, pages 144–153, 1998.
20. K. R. M. Leino and P. Müller. A verification methodology for model fields. In *ESOP'06*, volume 3924 of *LNCS*, pages 115–130. Springer, 2006.
21. K. R. M. Leino and G. Nelson. Data abstraction and information hiding. *ACM Trans. Prog. Lang. Syst.*, 24(5):491–553, 2002.
22. K. R. M. Leino, A. Poetzsch-Heffter, and Y. Zhou. Using data groups to specify and check side effects. In *PLDI'02*. ACM, 2002.
23. C. Morgan. *Programming from specifications (2nd ed.)*. Prentice Hall International (UK) Ltd., 1994.
24. M. Parkinson. Class invariants: The end of the road? In *IWACO'07*, 2007. `http://www.cs.purdue.edu/homes/wrigstad/iwaco/`.
25. J. C. Reynolds. Separation logic: a logic for shared mutable data structures. In *17h Annual IEEE Symposium on Logic in Computer Science*. IEEE Comp. Soc. Press, 2002.
26. A. Tafat, S. Boulmé, and C. Marché. A refinement approach for correct-by-construction object-oriented programs. Technical Report RR-7310, INRIA, 2010.
27. J.-P. Talpin and P. Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3):245–271, 1992.
28. M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997. Academic Press.
29. K. Zee, V. Kuncak, and M. Rinard. Full functional verification of linked data structures. In *PLDI'08*, pages 349–361. ACM Press, 2008.

Data refinement based testing

David Faitelson[1] and Shmuel Tyszberowicz[2]

[1] ProActive Modeling,
david@proactivemodeling.com
[2] The Academic College, Tel Aviv Yaffo, Israel
tyshbe@tau.ac.il

Abstract. FineFit is a model-driven framework for testing object-oriented systems. A FineFit specification is a collection of HTML tables representing the structure and the operations of the system under test. FineFit translates the specification into a relational model that serves both as a test oracle and as a source of test cases. FineFit uses the retrieve relation — the data refinement definition of the relationship between the abstract and the concrete system states — to check if the actual behavior of a Java program matches its abstract specification. The emphasis on representing and comparing system states makes this approach particularly attractive to object-oriented systems, which often consist of a complex graph of objects that represent the entities of a problem domain and the relationships between them. Thus, FineFit demonstrates the advantages of a data refinement approach to testing.

1 Introduction

Many object-oriented programs consist of a complex graph of objects that represent the entities of the problem domain and the relationships between them. We may say that these systems have a rich data model. To verify the correctness of such systems we must explicitly map the concrete representation of the data model to its abstract specification. Data refinement [3] is a theory that formally captures the relationship between an abstract specification and its concrete representation. In particular the notion of a *retrieve relation* formally captures the relationship between abstract and concrete representations of states. Data refinement is the underlying theory behind proof-oriented techniques [11, 14] that are used to develop safety critical systems [15, 7]. Unfortunately, proving data refinement is not practical for most commercial systems because they are built from existing off-the-shelf components for which no proofs of correctness exist. Additionally, the common languages in which we develop these systems have complicated semantics that make it very difficult to reason about formally.

However, we can use data refinement as an effective theory for testing. Instead of trying to prove that an implementation is a data refinement of its specification, we can use the specification as a test oracle (and, as we shall see, as a source of test cases) to automatically check if the behavior of the program satisfies the laws of data refinement. Yet, deducing the retrieve relation automatically from

the program is very difficult (indeed it brings back all the problems that are related with formal proofs in today's popular programming languages). Fortunately, we can delegate this task to the programmer — after all she must know how her data structures implement the system's specification. By delegating the implementation of the retrieve relation to the programmer we make the testing framework flexible and scalable because she can control which parts of the system to expose for the purpose of testing, even if the system is too large and complicated for automatic analysis.

To demonstrate the feasibility of this approach we have developed FineFit. FineFit is a proof of concept prototype that automatically tests Java programs for data refinement. FineFit was inspired by the following tools and ideas:

1. Data refinement [3] provides the correctness criteria and the theoretical foundation behind the testing algorithm.
2. Parnas tables [10] inspired us to use a tabular specification notation.
3. The Fit framework for integrated testing [9] gave us two important ideas:
   (a) Representing tables in HTML.
   (b) *code fixtures* that act as device drivers connecting the system under test (SUT) to the testing framework.
4. The Alloy analyzer [5] from which we have taken:
   (a) The relational semantics and the concrete syntax of the expressions that we write inside the tables.
   (b) The Kodkod relational constraint solver [12] to check for consistency, generate test cases, and check for data refinement compliance.

To test a system using FineFit we first write the system's specification as a collection of HTML tables (which may be generated from any tool that supports the HTML format, including Microsoft Word, Excel, Open-Office and Google docs) that define the structure and the operations of the SUT.

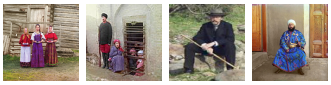
FineFit translates the system's specification into an Alloy relational model and ensures that it is internally consistent. Next, FineFit finds an operation that is available from the current system state, applies it to the SUT and checks if the new state corresponds to the operation's specification. This process continues until either FineFit finds a discrepancy or until we decide to stop the process. Figure 1 in Section 3.4 illustrates the testing process.

During testing, FineFit prints a trace that consists of the abstract snapshots (states) and the operation calls of the SUT. Thus, when FineFit detects a discrepancy we can review the entire history that led to the problem. This can greatly help us to investigate the reason for the problem.

The rest of the paper is organized as follows. In Section 2 we demonstrate FineFit using a concrete example. Then, in Section 3 we present FineFit in a general context and explain the essential ideas of its implementation. Following that, in Section 4 we describe the theory on which FineFit is based. In the last section we discuss FineFit's limitations, compare it to other related work and describe ideas for future work.

2 The case study

We illustrate FineFit with a case study taken from a commercial system that the first author is currently developing. We wish to test a simple photo album that consists of a sequence of photos which a user, Alice, can manipulate. Alice may append a photo at the end of the sequence, remove photos (from any place in the sequence) or replace photos by other photos. However, only the identifiers are stored in the album; the actual photos are stored in a remote server. When Alice is happy with her new album she presses 'save', at which point the difference between the old and the new state of the album is sent to the server. The difference consists of any new photos that were added to the album and the unique identifiers of existing photos that were deleted from the album. The server then stores the new photos and removes the deleted photos. For example, assume that we have the following photos with their identifiers:



PID_1 PID_2 PID_3 PID_4

The scenario below demonstrates what happens when Alice interacts with the photo album and how the variables that define the album's state are affected by the interaction. Each step in the scenario describes an action and shows the state of the system immediately after Alice performed that action:

	Action taken	**Album sequence** 1	2	3	**toAdd**	**existing**	**toDelete**
1	Alice already has two photos in her album. These photos are stored in the server.				{}	$\{PID_3, PID_1\}$	{}
2	Now Alice decides to add a new photo. The photo is placed into the first available position (3).				$\{PID_2\}$	$\{PID_3, PID_1\}$	{}
3	Next Alice remove the first photo. This causes the other photos to shift to the left.				$\{PID_2\}$	$\{PID_3, PID_1\}$	$\{PID_3\}$
4	Then Alice replaces the last photo with a new photo (PID_4).				$\{PID_4\}$	$\{PID_3, PID_1\}$	$\{PID_3\}$
5	Finally Alice presses save. As a result the server receives one new photo (PID_4) to add and one existing photo id (PID_3) to delete.				{}	$\{PID_1, PID_4\}$	{}

2.1 Modeling the photo album

To model the photo album using FineFit we must define four things:

1. the basic entities (atoms) that appear in the album,
2. the state of the album,
3. the album's invariant,
4. the operations of the album.

Atoms We use three basic entities to describe the album: photos, photo identifiers and integers (which index the photos in the album). To simplify the model we may assume that there is an injective mapping between photos and photo identifiers, and use only photo identifiers in the model. We therefore define two kinds of atoms:

Atom	Scope
PID	3
Int	3

Because FineFit uses a finite constraint solver the size of our model must be small. This is achieved by associating a scope with each atom that determines its maximal number of instances. In this example we have decided to test systems with at most three photo identifiers and three integers.

The album's state We can describe the state of the photo album by capturing the sequence of photo identifiers in the album and the content of the *toAdd*, *existing* and *toDelete* sets:

State	
album	seq PID
toAdd	set PID
existing	set PID
toDelete	set PID

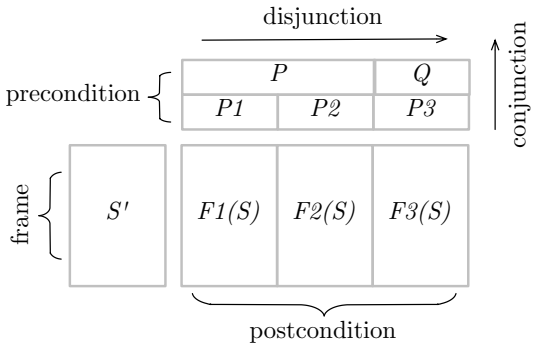
The album's invariant Of course, not every combination of values for the state's components represents a valid album. For example, a photo cannot be both new (i.e. it was not previously saved) and existing (i.e. it was previously saved). In addition, the sequence of photos must be injective because the album's business logic forbids the same photo from appearing more than once in the album. We use FineFit's invariant table to define the album's valid states:

Invariant	
`album in Int lone -> lone PID`	the album is an injective sequence
`no toAdd & existing`	we cannot add an existing photo
`toDelete in existing`	we can only delete existing photos
`no toAdd & toDelete`	cannot add and delete the same photo
`#album <= 3`	no more than 3 photos

The invariant is the conjunction of the rows in the table. We use Alloy's notation to write the constraints and the expressions. Briefly, every expression

is a relation (sets are relations of arity-1), the formula `x in y` stands for either membership or inclusion, depending on whether `x` is a single atom or a relation, the expression `X lone -> lone Y` represents the set of all injective functions from `X` to `Y`, the formula `no x` means that the relation `x` has no tuples.

The album's operations We define operations using operation tables. An operation table consists of three major parts: a precondition header, a frame column and a sequence of postcondition columns. The precondition is a predicate that captures the states and inputs from which we may apply the operation. The frame describes which parts of the state the operation may modify. Finally, the postcondition consists of a set of equations that define the new value of each component of the state as a function of the current state:



To understand the meaning of the precondition header it is best to read it from the bottom row towards the top row. The bottom row contains a separate *leaf predicate* for each case of the operation's behavior. Predicates that are common to a group of cases are factored into the row above. In the example we have three different cases P_1, P_2, and P_3. The predicate P, which is located above P_1 and P_2, represents a constraint that is common to both P_1 and P_2. There is a postcondition column for each leaf predicate. Each postcondition column contains an expression for each variable in the frame.

The following table is the complete specification of the *addPhoto* operation[3]:

	`#album < 3`			`#album = 3`
	`pid? !in toAdd + existing`	`pid? in toDelete`	`pid? in album.elems`	`true`
`album`	`album.add[pid?]`	`album.add[pid?]`	`album`	`album`
`toAdd`	`toAdd + pid?`	`toAdd`	`toAdd`	`toAdd`
`toDelete`	`toDelete`	`toDelete - pid?`	`toDelete`	`toDelete`
`report!`	`OK`	`OK`	`PHOTO_EXISTS`	`ALBUM_FULL`

[3] The operators `+`,`-`,`&` stand for union, set-difference, and intersection respectively; the expression `xs.add[y]` appends `y` at the end of the `xs` sequence; the expression `xs.elems` denotes the set of elements held in `xs`; the suffixes `?` and `!` denote respectively input and output variables.

The leftmost column contains the operation's frame — the parts of the state that the operation may change. The top two rows contain the operation's precondition. The precondition is divided into two major cases: the album is not full or the album is full. The first major case is further divided into three sub cases: the input photo is not already in the album, the input photo was previously deleted or the input photo already exists in the album. The column below each case represents the operation's postcondition in this case. Each row in the postcondition column corresponds to the same row in the frame. For example, the first column has the following meaning: if the album is not yet full and the input photo is not already in the album then add the new photo to *album* and to *toAdd* and set *report*! to *OK*. The content of *toDelete* is not changed. The *existing* set does not appear in the frame because it is not changed.

2.2 Implementing the photo album

The implementation of the photo album consists of two Java classes. The *Photo* class consists of a status that says if the photo is new or existing (stored in the server), and a string[4] that holds the content of the image. The *PhotoAlbum* class consists of a list of photos that defines the order of the photos in the album, and a set that contains the photo identifiers of the photos that we should delete:

```
public class Photo {
  public enum
     Status {Exists, New};
  Status status;
  String image;

  // operations ...
}
```

```
public class PhotoAlbum {
  List<Photo> album;
  Set<String> toDelete;

  // operations ...
}
```

Unfortunately we cannot list the entire implementation, but the program's classes demonstrate that in practice there is no simple correspondence between the data structures of the specification and the SUT.

2.3 Testing the photo album

In order to test the album we must perform the following steps:

1. Connect the album's code to FineFit by writing a method (named `retrieve`) that translates the concrete state into an instance of the abstract state.
2. Run the tests by creating and running a program that consists of the FineFit library and the album's code (see also Section 3.3).

[4] To simplify the prototype we represent the image and the photo's identifier using the same string. In the actual system the photo's identifier is calculated as a hash of the bitmap image.

We will describe the first step in Section 3.2 because it requires more background on the structure of FineFit.

Testing FineFit first ensures that the specification is consistent. It then begins testing the SUT by finding which operations and inputs may be called from the current state, calling them and comparing their effect against the specification. As the test runs, FineFit prints a trace that consists of the operation's name and inputs and the abstraction of the SUT's state.

The trace represents the behavior of the SUT as seen through the retrieve function. Thus, when we detect an error we have at our disposal the entire sequence of operations and states that led to that error. This history is an invaluable tool for understanding and correcting errors. Here is a trace that ends with a state discrepancy — a difference between the behavior of the SUT and the specification:

```
1  $ java PhotoAlbumModel
2  System is consistent.
3      init ->
4  album = []
5  toAdd = []
6  existing = []
7  toDelete = []
8  ...
9      removePhoto.SeqIdx0 ->
10 album = [[SeqIdx0, PID1]]
11 toAdd = []
12 existing = [[PID0], [PID1], [PID2]]
13 toDelete = [[PID0], [PID2]]
14     addPhoto.PID2 ->
15 album = [[SeqIdx0,PID1],[SeqIdx1,PID2]]
16 toAdd = [[PID2]]
17 existing = [[PID0], [PID1]]
18 toDelete = [[PID0]]
19 STATE DISCREPANCY
20 $
```

After declaring that the model is consistent (line 2) FineFit initializes the SUT (line 3) and displays the resulting initial state (lines 4-7). The trace then proceeds in the same fashion. After 30 calls, FineFit stops with a state discrepancy error (line 19). The error is that after PID_2 was deleted and then added back again (line 14), it disappeared from *existing* (compare lines 12 and 17). In this case the culprit is a bug in the implementation. When we add a previously deleted photo, we mark it (mistakenly) as new.

3 FineFit

The FineFit testing framework consists of the following parts:

- A tabular modeling notation in which we specify the SUT.
- A relational constraint solver that we use for four purposes:
  1. to check the consistency of the specification,
  2. to calculate which operations are available from a given state,
  3. to generate inputs for an operation we would like to apply,
  4. to check for violations of data refinement.

In addition, the SUT must implement the following Java interface which FineFit calls during the testing process:

```
public kodkod.Instance retrieve(kodkod.Universe universe);
```

This method takes the universe of atoms of the abstract model and returns the abstraction of the current SUT's state constructed as a set of tuples from the atoms in the universe. We show an example in Section 3.2.

3.1 The Kodkod relational constraint solver

Kodkod [12] is a Java library that implements a bounded relational constraint solver. It is the engine on which the Alloy analyzer [5] is currently implemented. Kodkod translates first order constraints over bounded relational terms into a SAT problem, applies a SAT solver to the problem and translates the solutions (if any) back into relational models. The Kodkod class library consists of the following major classes:

Atom an uninterpreted object (often just strings) that represents the basic entities of a specification.
Universe a set of atoms.
Tuple a sequence of atoms taken from a particular universe.
Expression either a relation or a relational expression such as union, intersection, join and so on.
Formula a first order logic constraint over expressions. For example, that one relation is a subset of another relation.
Bounds an association of relations to the power set of all the tuples they can take in a particular universe.
Instance an association of relations to particular sets of tuples.

In order to use Kodkod we follow these steps:

1. Create a universe and populate it with atoms. In our case study we create two kinds of atoms, one for photo identifiers and one for integers.
2. Create the specification:
   (a) Create the relations that describe the system. For example, a unary relation *PID* that represents the set of all possible photo identifiers and a binary relation *toAdd* that associates every state with the set of new photos in this state.
   (b) Create a formula that defines relationships between the relations. For example, a formula that insists that in every state the intersection of *toAdd* and *existing* is empty.
3. Create bounds for the relations by associating them with power sets of tuples. For example, the *PID* relation is associated with the power set of the atoms PID_1, PID_2, PID_3.
4. Create a Kodkod solver, giving it the formula and the bounds object.
5. Ask the solver to find a solution, that is, to find from the set of all possible bindings a particular subset that satisfies the formula.

If the solver finds a solution it returns an instance that contains the satisfying bindings, otherwise it returns an error code indicating that no solution exists.

3.2 Connecting the SUT to FineFit

Before FineFit can test the program we must implement the retrieve method that translates the concrete program state into an instance of the abstract state[5]. In our example this means that we have to translate the objects in the `album` and `toDelete` containers into the equivalent set of tuples for each of the relations *album*, *toAdd*, *existing* and *toDelete*:

```
1  public Instance retrieve(Universe universe) {
2    TupleFactory factory = universe.factory();
3    Instance instance = new Instance(universe);
4
5    List<Tuple> albumTuples = new ArrayList();
6    List<Tuple> toAddTuples = new ArrayList();
7    List<Tuple> existingTuples = new ArrayList();
8    int i = 0;
9    for (Photo p : album) {
10       albumTuples.add(factory.tuple(i, p.getImage()));
11       if (p.getStatus() == Photo.Status.New)
12           toAddTuples.add(factory.tuple(p.getImage()));
13       else if (p.getStatus() == Photo.Status.Exists)
14           existingTuples.add(factory.tuple(p.getImage()));
15       ++i;
16   }
17   List<Tuple> toDeleteTuples = new ArrayList();
18   for (String key : toDelete) {
19       toDeleteTuples.add(factory.tuple(key));
20   }
21   instance.add(Relation.binary("album"), factory.setOf(albumTuples));
22   // ... similarly for the other relations
23
24   return instance;
25 }
```

In lines 5-7 we create temporary lists to hold the tuples of each relation and then (lines 9-21) traverse, first the `album` (lines 9-16), then the `toDelete` set (lines 18-21) and collect the appropriate tuples into the temporary lists. Finally (lines 21-22) we associate each relation with its list of tuples and return the result as a Kodkod `Instance` object (line 24).

3.3 Detecting model inconsistencies

There is no more reason to believe that we can write error-free specifications than there is to believe that we can write error-free programs. Therefore it is important that we ensure that our specification is consistent before we start testing. With the help of Kodkod we check two things:

[5] As we have mentioned in Section 1, we delegate the responsibility of implementing the retrieve relation to the programmer.

1. That there exists at least one state that satisfies the system's invariant. Formally, we ask Kodkod to find a solution to the following constraint:

$$\text{some } s : State \mid inv[s]$$

   If no model can satisfy this constraint then either the scope is too small or we have a contradiction in the invariant. In both cases this means that the specification has errors and cannot be used for testing.
2. That (at least within the given scope) every operation that starts from a valid state (one that satisfies the invariant) ends in a valid state. Formally, we ask Kodkod to find a solution to the negation of this requirement:

$$\text{some } s, s' : State \mid inv[s] \text{ and } op[s, s'] \text{ and } !\, inv[s']$$

If Kodkod can find a solution then there is a valid state from which the operation *op* takes the system to an invalid state. This means that the specification has errors and must be fixed before it can be used for testing. Once we have ensured that the specification is consistent we can begin to test the SUT.

3.4 The testing procedure

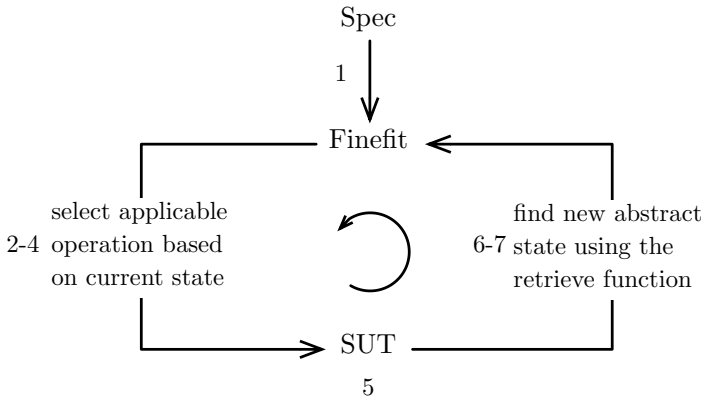


Fig. 1. A schematic view of the testing procedure. The numbers refer to the steps of the testing procedure. Steps 2 to 7 repeat in a loop until either FineFit finds a discrepancy or the user decides to stop.

The testing procedure (illustrated in Figure 1) uses two variables, *currentState* and *nextState*, to keep track of the SUT's state before and after each operation. The procedure consists of the following steps:

1. Initialize the SUT and set *currentState* to its initial state.

2. Use Kodkod to find all the operations that may be applied from *currentState*.
3. If no operation can be applied then stop and report a deadlock error.
4. Otherwise, select at random[6] one of the available operations, ask the SUT to apply this operation and set *nextState* to the new SUT's state.
5. Use Kodkod to check that the pair $(currentState, nextState)$ satisfies the operation's specification.
6. If the pair does not satisfy the specification then stop and report a data refinement error.
7. Otherwise, set *currentState* to *nextState* and go to step 2.

4 Data refinement

4.1 Systems as state based ADTs

Following [3] we represent a software system as a single abstract data type (ADT) that consists of a set of states S and a collection of operations OP indexed over the finite set I:

$$ADT = (S, (OP_i)_{i \in I})$$

Each state consists of named components (the state variables) that are mapped to some arbitrary values. Let V be a set of possible values and N be a finite set of names. The state space of the system, Σ, is the set of all total functions from N to V, that is $\Sigma = N \rightarrow V$. The set of system states, S, is a subset of Σ, that is $S \subseteq \Sigma$. Each operation is a relation between system states:

$$\forall\, i : I \bullet OP_i \in S \leftrightarrow S$$

We say that the ADT $(S, (OP_i)_{i \in I})$ is consistent if two conditions hold:

1. The set of system states is not empty:

$$S \neq \emptyset$$

2. For every operation OP_i, if the operation moves the system from state s to state s' and s is a valid system state, then s' must also be a valid system state:

$$\forall\, s : S;\ s' : \Sigma \bullet (s, s') \in OP_i \Rightarrow s' \in S$$

We can use different ADTs to describe the same system, depending on their abstraction level. The difference in the abstraction level is both in the states and in the operations of the ADT: a more abstract ADT will have states with less details (each state will have fewer components) than a more concrete ADT. In addition, a more abstract ADT will have operations that are less deterministic than a more concrete ADT.

Data refinement formally captures what it means for a concrete ADT to implement a more abstract ADT. The general idea is that for a concrete ADT C to behave according to an abstract ADT A, each operation of C must simulate its corresponding abstract operation. In the rest of this section we will explain precisely what this means.

[6] See also Section 5.4.

4.2 Forward simulation

To say that a concrete operation COP_i behaves according to an abstract operation AOP_i, we first have to explain what is the relationship between concrete and abstract states. Because a concrete state contains more details then an abstract state, it is natural to think of the abstract state as being encoded in the concrete state, and that therefore we should have a way of retrieving the abstract state from the concrete state. A retrieve relation is a relation between concrete and abstract states that defines how abstract states are retrieved from concrete states. Given two ADTs A and C that represent the same system, and a retrieve relation R that associates concrete states in C to their corresponding abstract states in A, we say that COP_i is a forward simulation of AOP_i if the following two conditions hold:

1. Every abstract state in AOP_i's domain has a corresponding concrete state:

$$\forall\, a : \mathrm{dom}\, AOP_i \bullet \exists\, c : \mathrm{dom}\, COP_i \bullet (c, a) \in R$$

2. For every concrete transition (c, c'), if the initial concrete state c corresponds to an initial abstract state a, then there exists an abstract transition (a, a') where the final abstract state a' corresponds to the final concrete state c':

$$\forall\, a : \mathrm{dom}\, AOP_i;\ c, c' : S_C \bullet (c, c') \in COP_i \wedge (c, a) \in R \Rightarrow \\ \exists\, a' : S_A \bullet (a, a') \in AOP_i \wedge (c', a') \in R$$

We can simplify this condition under two useful assumptions:

1. the concrete ADT is deterministic (i.e., its operations are functions),
2. the retrieve relation is a function.

In such a case the second condition of forward simulation simplifies to:

$$\forall\, a : \mathrm{dom}\, AOP_i;\ c, c' : S_C \bullet c \in \mathrm{dom}\, COP_i \wedge c \in \mathrm{dom}\, R \Rightarrow \\ COP_i(c) \in \mathrm{dom}\, R \wedge (R(c), R(COP_i(c)) \in AOP_i$$

In practice the first assumption applies to virtually all the cases where the concrete ADT is a program, and in such cases the second assumption is good software engineering. It would be very confusing if a single concrete state can be interpreted as different abstract states.

4.3 Testing for data refinement

Assume now that we have a concrete program C that is currently in a particular valid state c. Assume also that c can be mapped to a valid abstract state a of an abstract operation OP_A. We can check that the concrete operation OP_C behaves as specified by OP_A by using the following procedure:

1. Apply OP_C to the concrete system.

2. If the operation results in a crash we have found a problem. Either the precondition of OP_A is too weak (a bug in the specification) or there is a problem in the implementation OP_C.
3. Otherwise, the operation terminates successfully in the state $c' = OP_C(c)$.
4. If c' is not in the domain of R then we have found a problem. Either there's a mistake in OP_C or there's a mistake in R.
5. Otherwise, we can check if $(R(c), R(c'))$ is a valid transition of OP_A. If the check fails, then we have found a problem. Either the abstract operation or the concrete operation are wrong.
6. Otherwise, we can continue the test by taking c' to be the new current concrete state. Note that we already know that this state is in R's domain.

5 Discussion

5.1 Limitations

Currently, we provide only a limited way to specify non deterministic operations by defining overlapping cases in the operation's precondition. However this is only a problem of the current notation which we plan to address in the future.

Two more fundamental limitations are due to FineFit's underlying constraint solver: it cannot generate test cases for large data structures and it is not suitable for numerical specifications. The reason is that the more atoms we have in the model, the more time (and memory) it takes for Kodkod to find solutions.

Note however that the fact that FineFit cannot generate large test cases (that is, data structures that contain many objects) does not mean that it is not suitable for testing large systems (that is, systems that consist of many complicated components). On the contrary, as others have already noted [5], many kinds of errors can be illustrated on a small example and therefore will be detected by FineFit's testing strategy.

5.2 Performance

Even though it is possible to create models on which Kodkod performs badly (SAT is NP-complete after all), in our experience Kodkod is quite fast as long as the scope is small. For scopes of between 3 and 7 instances, the speed of analyzing the consistency of the model and of finding solutions using the sat4j SAT solver is on the order of a few tens of milliseconds on a 2.16 GHz Intel Core 2 Duo iMac.

As the size of the scope grows, the performance of course degrades exponentially. However, there are several arguments why we should prefer a small scope. First, even a small scope often generates a huge number of combinations which means that we will be exploring a very large state space (certainly much larger than anything we can hope to achieve by hand). Second, as a good testing practice we should use the smallest possible scope that can exhibit the SUT's behavior, because this makes it much easier to understand the problems that the

tests reveal. Finally, there is empirical evidence [1] to support Daniel Jackson's Small Scope Hypothesis: "Most bugs have small counterexamples" [5]. Thus, if we focus on the small test cases we are likely to find most of the bugs quickly and we will have a better chance at understanding them.

5.3 Related work

The idea of generating tests and a test oracle from a specification is not new. In this section we compare FineFit to related works.

Fit [9] is an open source tool for enhancing collaboration in software development. Fit automatically compares customers' expectations to actual results. It allows customers and testers to give examples of how a program should behave by writing these examples in HTML tables. Fit automatically checks those examples against the actual program. Each Fit table is interpreted by a *fixture* — a piece of code that is responsible for executing the SUT against the examples in the tables. Fit provides a very simple (and therefore effective) platform independent testing framework. However, the test cases must be derived manually from the specification. This is a laborious and error prone process that must be repeated every time the specification changes.

Parnas Tables [10] are tabular constructs that organize mathematical expressions, where rows and columns separate an expression into cases and each table entry specifies either the result value for some case or a condition that partially identifies some case. The tabular notation of FineFit is based on a combination of ideas taken from Fit and from Parnas tables. However, the specific details of FineFit's tabular notation are different from both tools. To the best of our knowledge the structure and semantics of FineFit's operation table is original and does not appear in previous works.

TestEra [6] is a framework for automated testing of Java programs that generates all non-isomorphic test cases within a given input size, and evaluates them against a correctness criteria. As an enabling technology, TestEra uses Alloy, and the Alloy Analyzer. *Korat [8]* has a similar purpose but it does not use Alloy and instead uses its own unique solver for expressing structural invariants. Both tools assume that the SUT can change its state to any arbitrary state that they choose. Unfortunately for many commercial systems this approach is not practical because the state of the component is often related to the state of other components forcing us to change the state of the entire system just to test a particular unit. In contrast, FineFit requires only that the component can *report* its current state, which is much easier for the concrete system to support. This makes FineFit more suitable than TestEra or Korat for testing large systems.

Spec Explorer [13] is a platform for writing model programs and using them to verify and test reactive object-oriented systems. Spec explorer provides tools for generating and visualizing test scenarios and for executing them against the SUT. The main difference between FineFit and Spec Explorer is that in Spec Explorer the state is an opaque object (often identified with a simple label whereas in FineFit a state has an internal structure and is more similar to an actual program state. Thus, Spec Explorer is better suited for testing reactive

systems while FineFit is better suited for testing data processing system where as the system moves from state to state we are more interested in the evolution and integrity of the data structures in each state.

Verification based testing (VBT) is a technique for generating test cases from correctness proofs. In [4] this idea is implemented using the KeY verification system [2]. KeY is intended for verifying the correctness of security-critical Java and Java Card programs. The idea is that the structure of a proof tree corresponds to the possible execution paths of the program. It is then possible, by using a constraint solver, to calculate the test data that will force the program to follow a particular execution. The benefit of a VBT approach is that it can guarantee the coverage of the code (given suitable coverage criteria). However, the drawbacks of VBT are that it requires a formal semantics of the programming language and that the test oracle is platform specific. Unfortunately, many popular languages have a complicated semantics that changes as they evolve. As a result we must constantly update the VBT framework to support the changes in the language. We must also implement a VBT framework for each new language. Finally, when we change the language (perhaps because we have to port the program to a different platform) we must rewrite its specification. In contrast, a FineFit model remains the same regardless of the SUT's platform.

5.4 Future work

The FineFit framework is currently under development. We have implemented its essential core and demonstrated that the approach works in practice. However, we have not yet completed its user interface. Once completed, we plan to release FineFit as an open source project. In the rest of this section we discuss the research ideas that we would like to explore in future versions.

User defined exploration heuristics Currently FineFit decides which operation to apply by picking at random one that is applicable in the current state. However, in many cases it can be more effective to guide this process. For example, we may use a Markov chain to define the probabilities of operations according to the system's state. This way we can create behavioral profiles that match the behavior of actual users or focus the exploration to particular areas of the application. We would like to implement a user defined Markov chain model in a future version of FineFit.

Testing reactive systems Reactive systems respond to events that arrive from its environment. That is, the tester is no longer the only entity that can change the system's state. We would like to extend FineFit to support testing of reactive systems.

Black boxes Often when a system crashes the reason is buried somewhere in its past. If we record the trace of the last N states and operations in a black box (either a log file or an area of the memory that we can retrieve from a core file when the program crashes), we can later replay the trace on the specification to find the first point at which the system misbehaved. We plan to add a replay feature to FineFit in a future version.

Supporting different languages Currently FineFit supports the testing of Java programs. However, by implementing the framework as a client/server system and defining a platform independent notation (for example XML or JSON) to represent atoms and tuples we can support languages other than Java.

References

1. A. Andoni, D. Daniliuc, S. Khurshid, and D. Marinov. Evaluating the "small scope hypothesis". Technical report, In POPL'02, 2002.
2. B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer, 2007.
3. W. de Roever and K. Engelhardt. *Data Refinement: model-oriented proof methods and their comparison*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1998.
4. C. Engel and R. Hähnle. Generating unit tests from formal proofs. In *Tests and Proofs (TAP)*, pages 169–188, 2007.
5. D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, 2006.
6. S. Khurshid and D. Marinov. TestEra: specification-based testing of Java programs using SAT. *Automated Software Engineering*, 11(4):403–434, 2004.
7. T. Lecomte, T. Servat, and G. Pouzancre. Formal methods in safety-critical railway systems. In *Brazilian Symposium on Formal Methods*, 2007.
8. A. Milicevic, S. Misailovic, D. Marinov, and S. Khurshid. Korat: A tool for generating structurally complex test inputs. In *the International Conference on Software Engineering*, pages 771–774. IEEE, 2007.
9. R. Mugridge and W. Cunningham. *Fit for Developing Software: framework for integrated tests*. Prentice Hall, 2005.
10. D. L. Parnas. Tabular representation of relations. CRL Report 260, Research Institute of Ontario (TRIO), McMaster University, 1992.
11. K. Robinson. The B method and the B toolkit. In *Algebraic Methodology and Software Technology*, pages 576–580. Springer, 1997.
12. E. Torlak and D. Jackson. Kodkod: A relational model finder. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4424 of *LNCS*, chapter 49, pages 632–647. Springer, 2007.
13. M. Veanes, C. Campbell, W. Grieskamp, W. Schulte, N. Tillmann, and L. Nachmanson. Model-based testing of object-oriented reactive systems with Spec Explorer. In *Formal Methods and Testing*, volume 4949 of *LNCS*, pages 39–76. Springer, 2008.
14. J Woodcock and J Davies. *Using Z: Specification, Refinement, and Proof*. Prentice Hall, 1996.
15. J. Woodcock, S. Stepney, D. Cooper, J. Clark, and J. Jacob. The certification of the Mondex electronic purse to ITSEC level E6. *Formal Aspects of Computing*, 2007.

Satisfiability Solving and Model Generation for Quantified First-order Logic Formulas

Christoph D. Gladisch*

University of Koblenz-Landau
Department of Computer Science
Germany

Abstract. The generation of models, i.e. interpretations, that satisfy first-order logic (FOL) formulas is an important problem in different application domains, such as, e.g., formal software verification, testing, and artificial intelligence. Satisfiability modulo theory (SMT) solvers are the state-of-the-art techniques for handling this problem. A major bottleneck is, however, the handling of quantified formulas.
Our contribution is a model generation technique for quantified formulas that is powered by a verification technique. The model generation technique can be used either stand-alone for model generation, or as a precomputation step for SMT solvers to eliminate quantifiers. Quantifier elimination in this sense is sound for showing satisfiability but not for refutational or validity proofs. A prototype of this technique is implemented.

1 Introduction

Showing the satisfiability of a first-order logic (FOL) formula means to show the existence of an interpretation in which the formula evaluates to true. This is an important and long studied problem in different application domains such as formal software verification, software testing, and artificial intelligence. In software verification and testing the models, i.e. interpretations, are used as counter examples to debug programs and specifications and to generate test data respectively.

Satisfiability modulo theory (SMT) solvers are the state-of-the-art techniques for showing satisfiability of FOL formulas and to generate models for FOL formulas. A major bottleneck is, however, the handling of quantifiers (see, e.g., [7, 19, 11, 20]). Quantifiers often lead to problems that are not in the decidable fragments of SMT solvers. In such cases an SMT solver returns the result *unknown*, which means that the solver cannot determine if the formula is satisfiable or not.

We propose a model generation technique that is not explicitly restricted to a specific class of formulas and which can therefore solve more general formulas than SMT solvers can solve. As a motivating example, assume we want to show the satisfiability of the formula

$$\forall x.(x \geqslant 0 \rightarrow prev(next(x)) = x) \tag{1}$$

* gladisch@uni-koblenz.de

where *prev* and *next* are uninterpreted function symbols. Some state-of-the-art SMT solvers — concretely we have tested Z3 [6], CVC3 [1], Yices [10, 9] — are in contrast to the proposed technique not capable to solve this formula. The reason is that this formula is not in the decidable fragment of the solvers because it combines arithmetics, uninterpreted functions, and quantification.

The proposed technique is also capable of generating only partial interpretations that satisfy only the quantified formulas, and return a residue of ground formulas that is to be shown satisfiable. In this mode the technique acts as a precomputation step for SMT solvers to eliminate quantifiers. Quantifier elimination in this sense is sound for showing satisfiability but not for refutational or validity proofs. However, for handling of quantifiers in refutational and validity proofs powerful instantiation based techniques already exist. These can be combined with the proposed technique in order to create semi-decision procedures.

While model generation is not a new idea, the novelty of our approach are (1) the choice of language to represent (partial) interpretations, (2) the technique for construction of models, and (3) the means to evaluate (quantified) formulas under these interpretations. Since satisfiability solving and model generation for ground formulas is already well studied, we concentrate on the handling of quantified formulas.

Furthermore we would like to motivate the importance of satisfiability solving for software verification and testing. Software verification is a costly task mainly because programs and specifications have bugs and also because annotations such as invariants are often too weak to show desired program properties. We experience that during software verification most time is spend with fixing and adjusting the programs, specifications, and annotations. It is therefore invaluable to detect if verification conditions have counter examples. This requires, however, the ability to show the satisfiability of formulas that often contain quantifiers. The generation of counter examples is further important in counter example guided abstraction refinement (CEGAR) [4] and for checking the consistency, i.e. contradiction-freeness, of axiomatizations and of preconditions in specifications. Once a program is correct and annotations are strong enough a state-of-the-art verification tool can afterwards prove the correctness of the program usually automatically.

1.1 Background and Related Work

One has to distinguish between different quantifiers in different contexts, namely between those that can be skolemized and those that cannot be skolemized. For instance, in an attempt to show the validity of the formula $\forall x.\varphi(x)$, the variable x can be skolemized, i.e. replaced by a fresh constant, because all symbols of the signature are implicitly universally quantified in this context. When showing the validity of $\exists x.\varphi(x)$, then skolemization is not possible. In contrast, when showing satisfiability, then skolemization is allowed for $\exists x.\varphi(x)$ but not for $\forall x.\varphi(x)$. Thus, assuming the formulas being in prenex form, the tricky cases are the handling of (a) existential quantification when showing validity and (b) universal

quantification when showing satisfiability. In order to handle case (a) some instantiation(s) of the quantified formulas can be created *hoping* to complete the proof. Soundness is preserved by any instantiation. The situation in case (b) is, however, worse when using instantiation-based methods, because these methods are sound only if a complete instantiation of the quantified formula is guaranteed.

A popular instantiation heuristic is E-matching [19] which was first used in the theorem prover Simplify [8]. E-matching is, however, not complete in general. In general a quantified formula $\forall x.\varphi(x)$ cannot be substituted by a satisfiability preserving conjunction $\varphi(t_0) \wedge \ldots \wedge \varphi(t_n)$ where $t_0 \ldots t_n$ are terms computed via E-matching. For this reason Simplify may produce unsound answers (see also [17]) as shown in the following example.

$$\forall h.\forall i.\forall v.rd(wr(h,i,v),i) = v \tag{2}$$

$$\forall h.\forall j.0 \leqslant rd(h,j) \wedge rd(h,j) \leqslant 2^{32} - 1 \tag{3}$$

Formula (2) is an axiom of the theory of arrays and (3) specifies that all array elements of all arrays have values between 0 and $2^{32} - 1$. The first axiom is used to specify heap memory in [18]. Formula (3) seems like a useful axiom to specify that all values in the heap memory have lower and upper bounds, as it is the case in computer systems. However, the conjunction (2) $\wedge$ (3) is inconsistent, i.e. it is false, which can be easily seen when considering the following instantiation $[h := wr(h_0, k, 2^{32}), j := k]$, (see [18]). Simplify, however, produces a counter example for $\neg((2) \wedge (3))$, which means that it satisfies the $false$ formula (2) $\wedge$ (3). E-matching may be used for sound satisfiability solving when a complete instantiation of quantifiers is ensured. For instance, completeness of instantiation via E-matching has been shown for the Bernays-Schönfinkel class in [12]. An important fragment of FOL for program specification which allows a complete instantiation is the Array Property Fragment [3]. E-matching is used in state-of-the-art SMT solvers such as Z3 [6], CVC3 [1], Yices [10, 9], and others (see [5]). Formula (1) which is solvable with our technique is, however, neither in the Bernays-Schönfinkel class nor in the Array Property Fragment.

Another set of approaches for finding instantiations of quantified formulas is based on free-variables (see e.g. [14]). These approaches focus, however, on validity or respectively unsatisfiability proofs and not on satisfiability solving. More precisely, they don't guarantee a complete instantiation of quantifiers.

Satisfiability of a formula can be shown by weakening the formula with existential quantifiers and then showing its validity, instead of satisfiability. This idea is followed in [22] for proving the existence of a state that reveals a software bug. The approach uses free variables in order to compute instantiations of the existentially quantified variables.

Quantifier elimination techniques, in the *traditional* sense, replace quantified formulas by *equivalent* ground formulas, i.e. without quantifiers. Popular methods are, e.g., the Fourier-Motzkin quantifier elimination procedure for linear rational arithmetic and Cooper's quantifier elimination procedure for Presburger arithmetic (see, e.g., [13] for more examples). These techniques are, in contrast to the proposed technique, not capable of eliminating the quantifier in, e.g., (1).

Since first-order logic is only semi-decidable, equivalence preserving quantifier elimination is possible only in special cases. The transformation of formulas by our technique is not equivalence preserving. The advantage of our technique is, however, that it is not restricted to a certain class of formulas.

Finally, Finite Model Finding methods regard the finite domain version of the satisfiability problem in first-order logic. These methods were developed primarily in the '90ies and in some later work such as [23]. Our approach handles, however, also infinite domains.

Structure of the paper. In Section 2 the basic idea of our approach is explained. In Section 3 the underlying formalism of our approach is introduced. The main sections are Section 4 and 5 where the approach is described in more detail and where we identify the crucial problems that have to be solved. The solution to the problems described in Section 4 is given in form of a theorem and the soundness of the theorem is proved. In Section 6 we report on our preliminary experiments with our approach and provide conclusions and further research plans.

2 The Basic Idea of our Approach

The basic idea of our approach is to generate a partial FOL model in which a quantified formula that we want to eliminate evaluates to true. A set of quantified formulas can be eliminated, i.e. evaluated to true, by successive extensions of the partial model. This approach can be continued also on ground formulas to generate complete models. While this basic idea is simple, the interesting questions are: how to represent the interpretations, how to generate (partial) models, and what calculus is suitable in order to evaluate formulas under those (partial) interpretations.

The approach that we suggest is to use programs to represent partial models and to use weakest precondition computation in order to evaluate the quantified formulas to true. Weakest precondition is a well-known concept in formal software verification and symbolic execution based test generation. A weakest precondition $wp(p, \varphi)$, where p is a program and φ is a formula, expresses all states such that execution of p in any of these states results in states in which φ evaluates to true. Here, program states and FOL interpretations are understood as the same concepts. Our approach is to generate for a given quantified formula φ a program p such that the final states of p satisfy φ. Thus a technique for program generation is one of our contributions.

For example, in order to solve (1), we could generate the following program (assuming, e.g., JAVA-like syntax and semantics):

```
for(i=0;true;i++){ next[i]=new T(); next[i].prev=i; }      (4)
```

and compute the weakest precondition of (1) with respect to this program, i.e. $wp((4), (1))$. Using a verification calculus the weakest precondition of the quantified subformula can be evaluated to true. Thus, effectively the quantified formula

is eliminated and a partial interpretation represented in form of a program is obtained.

A typical programming language such as JAVA is, however, not *directly* suitable for this task because function and predicate symbols are usually not representable in such languages. A verification calculus may also require extensions because loops are usually handled by the loop invariant rule and the loop invariant may introduce new quantified formulas.

A language and a calculus that are suitable for our purpose exist, however, in the verification system KeY. The language consists of so-called *updates*. In the following sections we introduce this language and describe our technique for construction of updates that evaluate quantified formulas to true while minimizing the chance of introducing new quantified formulas.

3 KeY's Dynamic Logic with Updates

The KeY system [2, 16] is a verification and test generation system for a subset of JAVA. The system is based on the logic JAVA CARD DL, which is an instance of Dynamic Logic (DL) [15]. Dynamic Logic is an extension of first-order logic with modal operators. The ingredients of the KeY system that are needed in this paper are first-order logic (FOL) extended by the modal operators *updates* [21].

Notation. We use the following abbreviations for syntactic entities: $\boldsymbol{V}$ is the set of (logic) variables; $\boldsymbol{\Sigma^f}$ is the set of function symbols; $\boldsymbol{\Sigma^f_r} \subset \Sigma^f$ is the set of rigid function symbols, i.e. functions with a fixed interpretation such as, e.g., '0', 'succ', '+'; $\boldsymbol{\Sigma^f_{nr}} \subset \Sigma^f$ is the set of non-rigid function symbols, i.e. uninterpreted functions; $\boldsymbol{\Sigma^p}$ is the set of predicate symbols; $\boldsymbol{\Sigma}$ is the signature consisting of $\Sigma^f \cup \Sigma^p$; $\boldsymbol{Trm_{FOL}}$ is the set of FOL terms; $\boldsymbol{Trm}$ is the set of DL terms; $\boldsymbol{Fml_{FOL}}$ is the set of FOL formulas; $\boldsymbol{Fml}$ is the set of DL formulas; $\boldsymbol{U}$ is the set of updates; $\doteq$ is the equality predicate; and $=$ is syntactic equivalence. The following abbreviations describe semantic sets: $\boldsymbol{\mathcal{D}}$ is the FOL domain or universe; $\boldsymbol{\mathcal{S}}$ is the set of states or equivalently the set of FOL interpretations. To describe semantic properties we use the following abbreviations: $\boldsymbol{val_s(t)} \in \mathcal{D}$ is the valuation of $t \in Trm$ and $\boldsymbol{val_s(u)} \in \mathcal{S}$ is the valuation of $u \in U$ in $s \in \mathcal{S}$; $\boldsymbol{s} \vDash \boldsymbol{\varphi}$ means that φ is true in state $s \in \mathcal{S}$; $\vDash \boldsymbol{\varphi}$ means that φ is valid, i.e. for all $s \in \mathcal{S}$, $s \vDash \varphi$; and $\equiv$ is semantic equivalence.

Updates capture the essence of programs, namely the state change computed by a program execution. States and FOL interpretations are the same concept. An update changes the interpretation of symbols Σ^f_{nr} such as uninterpreted functions. Hence, updates represent partial states and can be used to represent (partial) models of formulas. The set Σ^f_r represents rigid functions whose interpretation is fixed and cannot be changed by an update.

For instance, the formula $(\{a := b\}a = c) \in Fml$, where $a \in \Sigma^f_{nr}$ and $b, c \in \Sigma^f$ consists of the (function) update $a := b$ and the *application* of the update modal operator $\{a := b\}$ on the formula $a = c$. The meaning of this *update application* is the same as that of the weakest precondition $wp(a := b, a = c)$, i.e.

it represents all states such that after the assignment $a := b$ the formula $a = c$ is true — which is equivalent to $b = c$.

Definition 1. *Syntax. The sets U, Trm and Fml are inductively defined as the smallest sets satisfying the following conditions. Let $x \in V$; $u, u_1, u_2 \in U$; $f \in \Sigma^f_{nr}$; $t, t_1, t_2 \in Trm$; $\varphi \in Fml$.*

- *Updates. The set U of updates consists of: neutral update ε; function updates $(f(t_1, \ldots, t_n) := t)$, where $f(t_1, \ldots, t_n)$ is called the* location term *and t is the* value term *; parallel updates $(u_1 \,||\, u_2)$; conditional updates $(\texttt{if}\ \ \varphi; u)$;and quantified updates $(\texttt{for}\ x; \varphi; u)$.*
- *Terms. The set of Dynamic Logic terms includes all FOL terms, i.e. $Trm \supset Trm_{FOL}$; and $\{u\}t \in Trm$ for all $u \in U$ and $t \in Trm$.*
- *Formulas. The set of Dynamic Logic formulas includes all FOL formulas, i.e. $Fml \supset Fml_{FOL}$; $\{u\}\varphi \in Fml$ for all $u \in U$ and $\varphi \in Fml$; sequents $\Gamma \Longrightarrow \Delta \in Fml$, where $\Gamma, \Delta \subset Fml$; and all $\varphi \in Fml$ are closed by quantifiers, i.e. φ has no free variables.*

A sequent $\Gamma \Longrightarrow \Delta$ is equivalent to the formula $(\gamma_1 \wedge \ldots \wedge \gamma_n) \rightarrow (\delta_1 \vee \ldots \vee \delta_m)$, where $\gamma_1, \ldots, \gamma_n \in \Gamma$ and $\delta_1, \ldots, \delta_m \in \Delta$. Sequents are normally, e.g. in [2] , not included in the set of formulas. However, in this work it is convenient to include them to the set of formulas as *syntactic sugar*.

Definition 2. *Semantics. We use the notation from Def. 1, further let $s, s' \in \mathcal{S}$; $v, v_1, v_2 \in \mathcal{D}$; $x, x_i, x_j \in V$; and $\varphi(x)$ and $u(x)$ denote a formula resp. an update with an occurrence of x.*

Terms and Formulas

- *$val_s(\{u\}t) = val_{s'}(t)$, where $s' = val_s(u)$*
- *$val_s(\{u\}\varphi) = val_{s'}(\varphi)$, where $s' = val_s(u)$*

Updates

- *$s = val_s(\varepsilon)$*
- *$s' = val_s(f(t_1, \ldots, t_n) := t)$, where $s' = s$ except the interpretation of $f^{s'}$ is changed such that $val_{s'}(f(t_1, \ldots, t_n)) = val_s(t)$*
- *$s' = val_s(u_1; u_2)$, there is s'' with $s'' = val_s(u_1)$ and $s' = val_{s''}(u_2)$*
- *$s' = val_s(u_1 \,||\, u_2)$. We define s' by the interpretation of terms t.*
  Let $v_0 = val_s(t)$, $v_1 = val_s(\{u_1\}t)$, and $v_2 = val_s(\{u_2\}t)$.

$$\text{If } v_0 \neq v_2 \text{ then } val_{s'}(t) = v_2 \text{ else } val_{s'}(t) = v_1.$$

- *$s' = val_s(\texttt{if}\ \ \varphi; u)$, if $val_s(\varphi) = true$ then $s' = val_s(u)$, otherwise $s' = s$.*
- *Intuitively, a quantified update $(\texttt{for}\ x; \varphi(x); u(x))$ is equivalent to the infinite composition of parallel updates (parallel updates are associative):*

$$\ldots \,||\, (\texttt{if}\ \ \varphi(x_i); u(x_i)) \,||\, (\texttt{if}\ \ \varphi(x_j); u(x_j)) \,||\, \ldots$$

satisfying some global order $\succ$ such that $\beta(x_i) \succ \beta(x_j)$, where $\beta : V \rightarrow \mathcal{D}$.

A complete and formal definition of quantified updates cannot be given in the scope of this paper; we refer the reader to [21, 2] for a complete definition of the language and the simplification calculus. In the following some examples are shown of how updates, terms, and formulas are evaluated in KeY respecting the given semantics in Def 2.

- $\{f(1) := a\}f(2) = f(1)$ simplifies to $f(2) = a$.
- $\{f(b) := a\}P(f(c))$ simplifies to $(b \doteq c \rightarrow P(a)) \wedge (\neg b \doteq c \rightarrow P(f(c)))$.
- $\{f(a) := a\}f(f(f(a)))$ simplifies to a.
- $\{u_1; f(t_1, \ldots, t_n) := t\}$ is equivalent to $\{u_1 \,||\, f(\{u\}t_1, \ldots, \{u\}t_n) := \{u\}t\}$.
- $\{f(1) := a \,||\, f(2) := b\}f(2) = f(1)$ simplifies to $b = a$.
- $\{f(1) := a \,||\, f(1) := b\}f(2) = f(1)$ simplifies to $f(2) = b$, i.e. the last update *wins* in case of a conflict.
- $\{\texttt{if}\ \ \varphi;\ f(b) := a\}P(f(c))$ simplifies to $\varphi \rightarrow \{f(b) := a\}P(f(c))$.
- $\{\texttt{for}\ x;\ 0 \leqslant x \wedge x \leqslant 1;\ f(x) := x\}$ is equivalent to $\{f(1) := 1 \,||\, f(0) := 0\}$.

4 Model Generation by Iterative Update Construction

In order to show the satisfiability of a formula ϕ_{in}, our approach is to generate an update u, such that $\vDash \{u\}\phi_{in}$. If such an update exists, then ϕ_{in} is satisfiable and the update represents a model of ϕ_{in}.

Our main contribution is a technique for generating (partial) models for quantified formulas. As this work was developed in the context of KeY which is based on a sequent calculus, we regard the model generation problem of a quantified formula $\forall x.\phi(x)$ in a sequent $\varphi = (\Gamma, \forall x.\phi(x) \Longrightarrow \Delta)$. Such sequents occur frequently as open branches of failed proof attempts. The reason for proof failure is often unclear and it is desired to determine if φ has a counter example, i.e. if a model exists for $\neg\varphi$. The goal is therefore given by the following problem description.

Definition 3. *Problem Description. Given a sequent* $(\Gamma, \forall x.\phi(x) \Longrightarrow \Delta)$ *the goal is to generate an update* u *such that:*

$$(\{u\}(\Gamma, \forall x.\phi(x) \Longrightarrow \Delta)) \equiv (\Gamma', \mathit{true} \Longrightarrow \Delta') \tag{5}$$

If this problem is solved by a technique, then this technique can be applied iteratively to all quantified formulas occuring in Γ and Δ resulting in a sequent $\Gamma'' \Longrightarrow \Delta''$ that consists only of ground formulas. Note that non-skolemizable quantified formulas occuring in Δ are those with existential quantifiers and they can be *moved* to Γ using the following equivalence: $(\Gamma \Longrightarrow \exists x.\phi(x), \Delta) \equiv (\Gamma, \forall x.\neg\phi(x) \Longrightarrow \Delta)$.

We have implemented different algorithms that follow this approach. Unfortunately, only in rare cases the problem formulated in Def. 3 was solved by early algorithms. Based on experiments with early algorithms we have identified two important problems that we state in form of the following informal proposition.

Proposition 1. *The following description follows the notation of Def. 3.*

a) *In general cases of* $\forall x.\phi(x)$*, it is not feasible to construct an update* u *such that* $\vDash \{u\}\forall x.\phi(x)$*, without analysing the semantic properties of the matrix* $\phi(x)$*.*
b) *The theorem prover defined in [2] is not sufficiently powerful to simplify* $(\Gamma', \{u\}\forall x.\phi(x) \Longrightarrow \Delta')$ *to* $(\Gamma', true \Longrightarrow \Delta')$ *if* $\vDash \{u\}\forall x.\phi(x)$ *and* u *is a quantified update.*

Some possitilities to analyse the semantic properties of $\phi(x)$ are to test instances of $\phi(x)$ or to use free variables (see, e.g., [14]). We have experimented with the latter approach and could solve problem (a) in several cases. The reason for problem (b) is that in order to simplify the matrix $\phi(x)$ the sequent calculus requires semantic information about $\phi(x)$ to be available on the sequent level, i.e. in the formulas $\Gamma \cup \Delta$.

We have implemented an algorithm that solves both problems of Proposition 1. The algorithm itself is not provided in this paper; instead we provide a theorem that formalizes only the crucial problem simplification technique of the algorithm. The simplification technique is the core of the algorithm and we therefore prove the soundness of this simplification.

For the construction of the updates it is sometimes necessary to introduce and axiomatize fresh function symbols. For instance, it may be desired to introduce a fresh function $notZero \in \Sigma^f$ with the axiom $\neg(notZero \doteq 0)$. With this axiom it is, e.g., possible to write an update $a := b + notZero$, with $a, b \in Trm_{FOL}$, expressing a general assignment to a with a value different from b. Each update u_i is therefore associated with an axiom α_i.

Definition 4. *Given a sequent* $\varphi = (\Gamma, \underline{\forall x.\phi(x)} \Longrightarrow \Delta)$*, where* $\Gamma, \Delta \subset Fml$ *and* $\phi(x)$ *is an arbitrary formula with an occurrence of* $x \in V$*, i.e.* ϕ *is not restricted to* $\phi \in \Sigma^p$*. The formulas* $\psi_m, \varphi'_m, \varphi_m \in Fml$*, for* $m \in \mathbb{N}$*, are defined recursively as:*

- $\varphi_0 = (\Gamma, \underline{\forall x.\phi(x)} \Longrightarrow \Delta)$ $\quad$ $\varphi_{m+1} = \{u_m\}(\alpha_m \rightarrow \varphi_m)$
- $\varphi'_0 = (\Gamma, \underline{true} \Longrightarrow \Delta)$ $\quad$ $\varphi'_{m+1} = \{u_m\}(\alpha_m \rightarrow \varphi'_m)$
- $\psi_0 = (\Gamma \Longrightarrow \underline{\forall x.\phi(x)}, \Delta)$ $\quad$ $\psi_{m+1} = \{u_m\}(\alpha_m \rightarrow \psi_m)$

Definition 4 describes an abstract search technique for a sequence of updates $u_m\,;\ldots;\,u_0$, $m \in \mathbb{N}$, for solving the problem of Def. 3. The updates $u_m\,;\ldots;\,u_0$ constitute the update u in Def. 3 and $\varphi_0 \equiv \varphi$ is the original sequent that is to be shown falsifiable. In the following theorem we assume $\gamma = \underline{\forall x.\phi(x)}$.

Theorem 1. *Let* $m \in \mathbb{N}$*,* $u_0, \ldots, u_m \in U$*;* $\alpha_0, \ldots, \alpha_m \in Fml$*; let* $\varphi = (\Gamma, \gamma \Longrightarrow \Delta)$ *and* $\psi_m, \varphi'_m, \varphi_m \in Fml$ *be defined according to Def. 4, then*

i. $\vDash \psi_m \leftrightarrow (\varphi'_m \leftrightarrow \varphi_m)$
ii. If there is $s_m \in \mathcal{S}$ *such that* $s_m \vDash \neg\varphi_m$*, then there exists* $s \in \mathcal{S}$ *with* $s = val_{s_m}(u_m;\ldots;u_1;\varepsilon)$ *and* $s \vDash \neg\varphi$*.*

The theorem describes under what condition a sequence (not sequent) of update and axiom pairs $(u_0, \alpha_0), \ldots, (u_m, \alpha_m)$ evaluates a quantified formula to *true*; and the theorem describes how this sequence represents a partial model.

Formula $\neg\varphi$ is the formula for which a model shall be generated. Statement (ii) of Theorem 1 states that if there is a model $s_m \in \mathcal{S}$ for a formula $\neg\varphi_m$, according to Def. 4, then from s_m a model for $\neg\varphi$ can be derived by evaluation of the updates $u_0, \ldots, u_m$. Hence, $\neg\varphi_m$ can be used to show the satisfiability of $\neg\varphi$.

For instance, let $\varphi \equiv (\neg a = b)$, then a suitable pair (u_0, α_0) to construct φ_1 is, e.g. $(a := b, true)$. In this case φ_1 has the form $\{a := b\}(true \rightarrow (\neg a = b))$ which can be simplified to $false$. Hence, any state $s_1 \in \mathcal{S}$ satisfies $s_1 \vDash \neg\varphi_1$ which implies that $\neg\varphi$ is satisfiable and a model $s \in \mathcal{S}$ for $\neg\varphi$ is $s = val_{s_1}(a := b)$. Note, that chosing an update is a heuristic, e.g. the pair $(b := a, true)$ or the pair $(a := 1 \,||\, b := 1, true)$ are also suitable candidates.

An important property of the statement for the construction of an update search procedure is that soundness of the statement is preserved by any pair (u, α). For instance, consider the pair $(a := 1 \,||\, b := 2, true)$ or the pair $(a := b, false)$. In both cases φ_1 evaluates to $true$. Hence, there is no $s_1 \in \mathcal{S}$ such that $s_1 \vDash \neg\varphi_1$ and therefore no implication is made regarding the satisfiability of φ.

Based on statement (i) an algorithm can be constructed for the generation of models for ground formulas. The challenge is, however, to generate a model that satisfies a quantified formula that cannot be skolemized. If ψ_m is valid then the model generation problem for $\neg\varphi_m$ can be replaced by the model generation problem for $\neg\varphi'_m$ because φ_m and φ'_m are equivalent. Considering Def. 4, the statement is interesting because in φ'_m the quantified formula is eliminated, i.e. it is replaced by true. Together with Statement (ii), $\neg\varphi'_m$ can be used to generate a model for $\neg\varphi$.

The problem is to check if $\varphi_m \equiv \varphi'_m$, which is a generalization of the problem in Def. 3. Theorem 1 states that the problem $\varphi_m \equiv \varphi'_m$ can be solved by a validity proof of ψ_m. This allows solving the problems described in Proposition 1 because the quantified formula in ψ_m occurs negated wrt. φ_m and can therefore be skolemized — note that $(\Gamma, \forall x.\phi(x) \Longrightarrow \Delta) \equiv (\Gamma \Longrightarrow \neg\forall x.\phi(x), \Delta)$. When ψ_m is skolemized, then it is (a) easy to analyse the semantics of $\phi(sk)$, where $sk \in \Sigma^f$ is the skolem function, and (b) the propositional structure of $\phi(sk)$ can be *flattened* to the sequent level which is necessary to simplify quantified updates. In this way both problems described in Proposition 1 are solved.

The approach can be generalized for the generation models for ground formulas by using the more general Def. 5 instead of Def. 4 in Theorem 1.

Definition 5. *Given a sequent $\varphi = (\Gamma, \gamma \Longrightarrow \Delta)$, where $\Gamma, \Delta \subset Fml$ and $\gamma \in Fml$, let the formulas $\psi_m, \varphi'_m, \varphi_m \in Fml$, for $m \in \mathbb{N}$, be defined recursively as follows:*

- $\psi_0 = (\Gamma \Longrightarrow \gamma, \Delta)$ $\quad$ $\psi_{m+1} = \{u_m\}(\alpha_m \rightarrow \psi_m)$
- $\varphi'_0 = (\Gamma, true \Longrightarrow \Delta)$ $\quad$ $\varphi'_{m+1} = \{u_m\}(\alpha_m \rightarrow \varphi'_m)$
- $\varphi_0 = (\Gamma, \gamma \Longrightarrow \Delta)$ $\quad$ $\varphi_{m+1} = \{u_m\}(\alpha_m \rightarrow \varphi_m)$

In the proof of Theorem 1 we use the following lemma.

Lemma 1. *Weakening Update. Let $u \in U$ and $\varphi \in Fml$. If $\vDash \varphi$, then $\vDash \{u\}\varphi$.*

Proof of Lemma 1. Since for any $s \in \mathcal{S}$, holds $s \vDash \varphi$, it is also the case for $s' = val_s(u)$ that $s' \vDash \varphi$ because $s' \in \mathcal{S}$. ■

Proof of Theorem 1. The proof of Theorem 1 is based on induction on m.
Induction Base ($m = 0$). (i) Validity of

$$(\underbrace{\Gamma \Longrightarrow \forall x.\phi(x), \Delta}_{\psi_0}) \leftrightarrow ((\underbrace{\Gamma, true \Longrightarrow \Delta}_{\varphi_0'}) \leftrightarrow (\underbrace{\Gamma, \forall x.\phi(x) \Longrightarrow \Delta}_{\varphi_0}))$$

can be shown by using propositional transformation rules. In the following we simplify $\varphi_0' \leftrightarrow \varphi_0$ and derive by equivalence transformations ψ_0.

$$\begin{array}{c}((\Gamma \wedge true) \rightarrow \Delta) \leftrightarrow ((\Gamma \wedge \forall x.\phi(x)) \rightarrow \Delta)\\ (\Gamma \rightarrow \Delta) \leftrightarrow ((\Gamma \wedge \forall x.\phi(x)) \rightarrow \Delta)\end{array}$$

$$\begin{array}{c}(\Gamma \rightarrow \Delta) \rightarrow ((\Gamma \wedge \forall x.\phi(x)) \rightarrow \Delta)\\ ((\Gamma \rightarrow \Delta) \wedge \Gamma \wedge \forall x.\phi(x)) \rightarrow \Delta\\ (\Delta \wedge \Gamma \wedge \forall x.\phi(x)) \rightarrow \Delta\\ (\Delta \wedge \Gamma) \rightarrow \Delta\\ \Delta \rightarrow \Delta\\ true\end{array} \qquad \begin{array}{c}((\Gamma \wedge \forall x.\phi(x)) \rightarrow \Delta) \rightarrow (\Gamma \rightarrow \Delta)\\ (((\Gamma \wedge \forall x.\phi(x)) \rightarrow \Delta) \wedge \Gamma) \rightarrow \Delta\\ ((\forall x.\phi(x) \rightarrow \Delta) \wedge \Gamma) \rightarrow \Delta\\ ((\forall x.\phi(x) \rightarrow \Delta) \wedge \Gamma) \rightarrow \Delta\\ ((\neg\forall x.\phi(x) \wedge \Gamma) \rightarrow \Delta) \wedge ((\Delta \wedge \Gamma) \rightarrow \Delta)\\ (\neg\forall x.\phi(x) \wedge \Gamma) \rightarrow \Delta\\ \Gamma \rightarrow (\forall x.\phi(x) \vee \Delta)\end{array}$$

Since $\varphi_0 = \varphi$ and $s = val_{s_0}(\varepsilon) = s_0$ statement (ii) is trivially true.
Induction Step ($m \geqslant 0$). (i) Assuming $\vDash \psi_m \leftrightarrow (\varphi_m' \leftrightarrow \varphi_m)$, we want to show $\vDash \psi_{m+1} \leftrightarrow (\varphi_{m+1}' \leftrightarrow \varphi_{m+1})$. If $\vDash \psi_m \leftrightarrow (\varphi_m' \leftrightarrow \varphi_m)$, then

$$\vDash \alpha_m \rightarrow (\psi_m \leftrightarrow (\varphi_m' \leftrightarrow \varphi_m)) \tag{6}$$

for any $\alpha_m \in Fml$. We use the equivalence

$$(A \rightarrow (B \leftrightarrow C)) \leftrightarrow ((A \rightarrow B) \leftrightarrow (A \rightarrow C))$$

to derive the following statement that is equivalent to (6)

$$\vDash ((\alpha_m \rightarrow \psi_m) \leftrightarrow ((\alpha_m \rightarrow \varphi_m') \leftrightarrow (\alpha_m \rightarrow \varphi_m))) \tag{7}$$

Due to Lemma 1 (*ii*), (7) implies

$$\vDash \{u_m\}((\alpha_m \rightarrow \psi_m) \leftrightarrow ((\alpha_m \rightarrow \varphi_m') \leftrightarrow (\alpha_m \rightarrow \varphi_m))) \tag{8}$$

that can be simplified by update propagation to

$$\vDash (\{u_m\}(\alpha_m \rightarrow \psi_m) \leftrightarrow (\{u_m\}(\alpha_m \rightarrow \varphi_m') \leftrightarrow \{u_m\}(\alpha_m \rightarrow \varphi_m))) \tag{9}$$

Statement 9 is equivalent to $\vDash \psi_{m+1} \leftrightarrow (\varphi_{m+1}' \leftrightarrow \varphi_{m+1})$.

(ii) Assume there is $s_{m+1} \in \mathcal{S}$ such that $s_{m+1} \vDash \neg\varphi_{m+1}$. By propagating the negation of $\neg\varphi_{m+1}$ to the inside of the formula, loosely speaking, we obtain the equivalent formula $\varphi_m^{\neg} \in Fml$ that can be recursively defined as

$$\varphi_0^{\neg} = \neg(\Gamma, true \Longrightarrow \Delta) \qquad \varphi_{m+1}^{\neg} = \{u_m\}(\alpha_m \wedge \varphi_m^{\neg})$$

Hence, $s_{m+1} \vDash \neg\varphi_{m+1}$ is equivalent to $s_{m+1} \vDash \varphi^{\neg}_{m+1}$ which is equivalent to $s_{m+1} \vDash \{u_m\}(\alpha_m \wedge \varphi^{\neg}_m)$. There is $s_m \in \mathcal{S}$ with $s_m = val_{s_{m+1}}(u_m)$ such that $s_m \vDash \alpha_m \wedge \varphi^{\neg}_m$ and therefore $s_m \vDash \varphi^{\neg}_m$. Since $\varphi^{\neg}_m$ is equivalent to $\neg\varphi_m$ we have $s_m \vDash \neg\varphi_m$. According to the induction hypothesis there exists $s \in \mathcal{S}$ with $s = val_{s_m}(u_m; \ldots; u_1; \varepsilon)$ such that $s \vDash \neg\varphi$. Because of $s_m = val_{s_{m+1}}(u_m)$, we conclude that if $s_{m+1} \vDash \neg\varphi_{m+1}$, then there exists $s \in \mathcal{S}$ with $s = val_{s_{m+1}}(u_{m+1}; u_m; \ldots; u_1; \varepsilon)$ such that $s \vDash \neg\varphi$. ■

5 Heuristics for Update Construction from Formulas

While Section 4 describes a general sound framework for model generation, in this Section we shortly describe some heuristics that we have implemented to construct concrete updates. In particular we give an intuition of how quantified updates can be constructed in order to satisfy quantified formulas. Important to note is that soundness of Theorem 1 is preserved by *any* sequence of update and axiom pairs. Hence, unsoundness cannot be introduced by any of the heuristics.

Definition 6. *Update Construction. Let $\gamma \in Fml_{FOL}$ be the currently selected formula for which a partial model is to be created and which is a subformula in a sequent $\varphi = (\Gamma, \gamma \Longrightarrow \Delta)$. Let $\psi = (\Gamma \Longrightarrow \gamma, \Delta)$ and $\varphi' = (\Gamma \Longrightarrow \Delta)$.*

The goal of update construction from the formula γ is to create a pair (u, α), with $u \in U$ and $\alpha \in Fml$, such that

- $\vDash \{u\}(\alpha \rightarrow \psi)$, *and*
- *there is some $s \in \mathcal{S}$ with $s \vDash \neg\{u\}(\alpha \rightarrow \varphi')$*

The sequent ψ is equivalent to ψ_0 and φ' is equivalent to φ'_0, according to Def. 5. In a model search algorithm each time a pair (u_m, α_m) is constructed, new formulas $\varphi'_{m+1}, \varphi'_{m+1}$, and ψ_{m+1} are generated according to Def. 5. These formulas must be simplified to φ, ψ and, φ', respectively, such that a new formula $\gamma \in Fml_{FOL}$ can be selected for update construction according to Def. 6. In the following subsections, case distinctions are made on the structure of γ.

5.1 Update Construction from Ground Formulas

Handling of Equalities. Assume $t_1, t_2 \in Trm_{FOL}$ are location terms (see Def. 1). If γ is of the form $t_1 = l$ or $l = t_1$, where l is a literal, then the pair $(t_1 := l, true)$ should be created because $\vDash \{t_1 := l\}(true \rightarrow (t_1 \doteq l \wedge l \doteq t_1)$. If γ is of the form $t_1 = t_2$, a choice has to be made between the pairs $(t_1 := t_2, true)$ and $(t_2 := t_1, true)$. Equality between terms can in some cases also be established, if the terms share the same top-level function symbol and have location terms as arguments. For instance, let $f(t_1), f(t_2) \in Trm_{FOL}$ and $f \in \Sigma^f$, then $\vDash \{u\}(\alpha \rightarrow f(t_1) = f(t_2))$ can be satisfied by the pair $(t_1 := t_2, true)$ or by $(t_2 := t_1, true)$.

Handling of Arithmetic Expressions. Let $t_1, t_2 \in Trm_{FOL}$ be arithmetic expressions composed of rigid and non-rigid function symbols. Several solutions exist to satisfy $\vDash \{u\}(\alpha \rightarrow t_1 \doteq t_2)$. Consider for instance the polynomial equation

$$2 * a + b * c = d - e$$

where $a, b, c, d, e \in \Sigma^f_{nr}$ are location terms. There are five most general updates evaluating this equation to true. These can be obtained by solving the polynomial equation for one of the location terms at a time. Our implementation enumerates those solutions during update search. An example for one of the solutions is $((a := (d - e - b * c)/2, true)$.

Handling of Inequalities. Let $t_1, t_2 \in Trm_{FOL}$ where t_1 is a location term. An inequation $t_1 \neq t_2$ can be satisfied, e.g., by the pair $(t_1 := t_2 + 1, true)$. A more general update is, however, $t_1 := t_2 + notZero$, where $notZero \in \Sigma^f$ is a fresh-symbol representing a value different from 0. This is where the axiom part of a pair comes into play. A general solution for the formula $t_1 \neq t_2$ is the pair $(t_1 := t_2 + notZero, \neg(notZero = 0))$. Inequations of the form $t_1 < t_2$ can be handled by introducing a fresh symbol $gtZero \in \Sigma^f_{nr}$ with the axiom $gtZero > 0$.

5.2 Update Construction from Quantified Formulas

Our approach to create models for quantified formulas is to generate quantified updates. For example, the quantified formula

$$\forall x. x > a \rightarrow f(x) = g(x) + x \tag{10}$$

is satisfiable in any state after execution of the quantified update

$$\texttt{for}\ x;\ x > a;\ f(x) := g(x) + x \tag{11}$$

i.e. $\vDash \{(11)\}(10)$. Notice the similar syntactical structure between (10) and (11). Another solution is

$$\texttt{for}\ x;\ x > a;\ g(x) := f(x) - x \tag{12}$$

for which holds $\vDash \{(12)\}(10)$. It is easy to see that a translation can be generalized for other *simple* quantified formulas. Furthermore, the heuristics and case distinctions described in Section 5.1 can be reused to handle different arithmetic expressions and relations. For instance the formula

$$\forall x. f(x) \geqslant x \rightarrow (g(x) < f(x))$$

evaluates to true after execution of any of the following updates (with axioms)

$$(\texttt{for}\ x;\ f(x) \geqslant x;\ g(x) := f(x) + gtZero\ ,\ gtZero > 0)$$
$$(\texttt{for}\ x;\ \neg(g(x) < f(x));\ f(x) := x - gtZero\ ,\ gtZero > 0)$$

Java + JML

```
1  /*@ public normal_behavior
2    @ requires next!=null && prev!=null && next!=prev
3    @  && (\forall int k; true ; 0<=next[k] && next[k] < prev.length)
4    @  && (\forall int l; 0<=l && l<next.length; next[l]==l);
5    @ ensures (\forall int j; 0<=j && j<next.length; prev[next[j]]==j);
6    @ assignable prev[*]; */
7  public void link(){
8   /*@ loop_invariant (\forall int x; 0<=x && x <= i; prev[next[x]]==x)
9                   && (0<=i && i<=next.length) ; modifies prev[*],i; @*/
10  for(int i=0;i<next.length;i++){  prev[next[i]]=i; }
11 }
```

Java + JML

Fig. 1. An example of a Java method (of class `MyCls`) with a JML specification that is not verifiable because the underlined formula should be $x < i$ instead of $x \leqslant i$

The KeY tool implements a powerful update simplification calculus for quantified updates. The calculus may in some cases introduce new quantified formulas. In such cases our approach has to be applied either recursively on the new quantified formulas or the heuristic has to choose different updates in a search procedure to prevent the introduction of new quantified formulas.

Finally, the initial example of the paper, i.e. Formula (1), can be solved by the following quantified update application which the KeY system simplifies to true.

$$\{(\texttt{for}\ x_1;\ x_1 \geqslant 0;\ next(x_1) := x_1);(\texttt{for}\ x_2;\ x_2 \geqslant 0;\ prev(next(x_2)) := x_2)\}(1)$$

6 Experiments, Conclusions, and Future Work

We have proposed a model generation approach for quantified first-order logic (FOL) formulas that is based on weakest-precondition computation. The language we propose for representing models is KeY's update language. The advantage of using updates is the possibility to express models for quantified formulas via quantified updates, and the availability of a powerful calculus for simplifying formulas with updates to FOL formulas. In particular, no loop invariants have to be generated in order to simplify quantified updates.

We have identified problems (Proposition 1) that occur, when the approach is implemented according to the *basic* description. Theorem 1 provides a solution to these problems. The theorem allows us to reformulate the basic model generation approach for quantified formulas into a semantically equivalent approach without the problems described in Proposition 1.

Based on Theorem 1 and Definitions 4 and 5 an algorithm for model generation can be derived. The technique can be used in two ways. On the one hand, it can be used as a precomputation step to SMT solvers by restricting the

$\forall x : \texttt{int}.(x \leqslant -1 \vee x \geqslant 1 + i_0 \vee \texttt{get}_0(\texttt{prev}(\texttt{self}), \texttt{acc}_{[]}(\texttt{next}(\texttt{self}), x) \doteq x),$
$\forall x : \texttt{MyCls}.(\texttt{prevAtPre}(x) \doteq \texttt{prev}(x)),$
$\forall x : \texttt{MyCls}.(x \doteq \texttt{null} \vee \neg\texttt{created}(x) \vee \neg\texttt{a}(x) \doteq \texttt{null}),$
$\forall x : \texttt{MyCls}.(x \doteq \texttt{null} \vee \neg\texttt{created}(x) \vee \neg\texttt{next}(x) \doteq \texttt{null}),$
$\forall x : \texttt{MyCls}.(x \doteq \texttt{null} \vee \neg\texttt{created}(x) \vee \neg\texttt{prev}(x) \doteq \texttt{null}),$
$\forall x : \texttt{int}.\texttt{acc}_{[]}(\texttt{next}(\texttt{self}), x) \geqslant 0),$
$\forall x : \texttt{int}.\texttt{acc}_{[]}(\texttt{next}(\texttt{self}), x) \leqslant -1 + \texttt{length}(\texttt{prev}(\texttt{self}))),$
$\forall x : \texttt{int}.(l \leqslant -1 \vee l \geqslant \texttt{length}(\texttt{next}(\texttt{self})) \vee \texttt{acc}_{[]}(\texttt{next}(\texttt{self}), x) \doteq x),$
$\ldots \Longrightarrow \ldots$

Fig. 2. Quantified formulas in a sequent resulting from a failed verification attempt of the code in Figure 1; 21 additional ground formulas are abbreviated by '...'

...
$\{\texttt{for}\ x : \texttt{MyCls};\ (\texttt{next}(x) \doteq \texttt{null} \wedge \neg\texttt{a}(x) \doteq \texttt{null} \wedge \ldots);\ \texttt{created}(x) := \texttt{false}\}$
$\{\texttt{for}\ x : \texttt{MyCls};\ (\texttt{a}(x) \doteq 0 \wedge \neg x \doteq \texttt{null});\ \texttt{created}(x) := \texttt{false}\}$
$\{\texttt{for}\ x : \texttt{int};\ (b \geqslant 1 + x \wedge x \leqslant -1);\ \texttt{acc}_{[]}(\texttt{next}(\texttt{self})) := -1 + c_2\}$
$\{\texttt{for}\ x : \texttt{int};\ x \leqslant -1;\ i := \texttt{acc}_{[]}(\texttt{next}(\texttt{self})) - c_0 * -1 + c_1\}$
$\{\texttt{for}\ x : \texttt{int};\ (x \geqslant 0 \wedge x \geqslant 1 + i_0);\ \texttt{acc}_{[]}(\texttt{next}(\texttt{self})) := \texttt{length}(\texttt{prev}(\texttt{self})) + c_0\}$
$\{\texttt{for}\ x : \texttt{int};\ (\texttt{acc}_{[]}(\texttt{next}(\texttt{self}), x) = x \wedge x \leqslant i_0 \wedge \ldots);\ \texttt{get}_0(\texttt{prev}(\texttt{self}), x) := x\}$

Fig. 3. A subset of generated updates satisfying the quantified formulas in Figure 2

computation of the formulas ψ_m, φ'_m, and φ_m to Def 4. In this case the technique eliminates quantified formulas and leaves a residue of ground formulas or alternative quantified formulas to be solved by a different method, e.g. an SMT solver. On the other hand, the technique can be used stand-alone for model generation by using the general Def. 5.

The approach was developed in the context of formal software verification and test generation project. Verification attempts often fail, i.e., they are interrupted by a timeout. Figure 1 shows a JAVA method with a JML specification. A verification attempt of the method results in a set of open proof obligations. One of them is shown in Figure 2 that we abbreviate as φ. For a verification engineer it is important to know if the open proof obligation has a counter example or not. State-of-the-art approaches use SMT solvers to try answering such questions. These are, however, not powerful enough to solve formulas such as φ. Preliminary experiments show that our method can generate counter examples for formulas such as φ that SMT solvers cannot solve. For instance, Figure 3 shows a part of an iterative update application that describes a model for $\neg\varphi$ and was generated by an implementation of our approach.

What formulas can be solved by our general approach depends on the chosen language for model representation, the theorem prover in use, and the heuristics for model construction. Quantified formulas are suitable to represent models for certain kinds of quantified formulas. They are, however, not sufficient to represent models of inductively defined functions. We are currently working on an extension of the update language for this purpose.

References

1. Clark Barrett and Cesare Tinelli. Cvc3. In *CAV*, pages 298–302, 2007.
2. Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer, 2007.
3. Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. What's decidable about arrays? In *VMCAI*, pages 427–442, 2006.
4. Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *CAV*, pages 154–169, 2000.
5. Leonardo Mendonça de Moura and Nikolaj Bjørner. Efficient e-matching for smt solvers. In *CADE*, pages 183–198, 2007.
6. Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *TACAS*, pages 337–340, 2008.
7. David Déharbe and Silvio Ranise. Satisfiability solving for software verification. *STTT*, 11(3):255–260, 2009.
8. David Detlefs, David Detlefs, Greg Nelson, Greg Nelson, James B. Saxe, and James B. Saxe. Simplify: A theorem prover for program checking. Technical report, J. ACM, 2003.
9. Bruno Dutertre and Leonardo de Moura. The YICES SMT solver. Technical report, Computer Science Laboratory, SRI International, 2006. `http://yices.csl.sri.com/tool-paper.pdf`.
10. Bruno Dutertre and Leonardo Mendonça de Moura. A fast linear-arithmetic solver for dpll(t). In *CAV*, pages 81–94, 2006.
11. Yeting Ge, Clark W. Barrett, and Cesare Tinelli. Solving quantified verification conditions using satisfiability modulo theories. *Ann. Math. Artif. Intell.*, 55(1-2):101–122, 2009.
12. Yeting Ge and Leonardo Mendonça de Moura. Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In *CAV*, pages 306–320, 2009.
13. Silvio Ghilardi. Quantifier elimination and provers integration. *Electr. Notes Theor. Comput. Sci.*, 86(1), 2003.
14. Martin Giese. Incremental closure of free variable tableaux. In *IJCAR*, pages 545–560, 2001.
15. David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logic*. The MIT Press, London, England, 2000.
16. KeY project homepage. At `http://www.key-project.org/`.
17. Joseph R. Kiniry, Alan E. Morkan, and Barry Denby. Soundness and completeness warnings in esc/java2. In *Proc. Fifth Int. Workshop Specification and Verification of Component-Based Systems*, pages pp. 19–24, 2006.
18. Michał Moskal. *Satisfiability Modulo Software*. PhD thesis, University of Wrocław, 2009.
19. Michał Moskal, Jakub Lopuszanski, and Joseph R. Kiniry. E-matching for fun and profit. *Electr. Notes Theor. Comput. Sci.*, 198(2):19–35, 2008.
20. Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. Challenges in satisfiability modulo theories. In *RTA*, pages 2–18, 2007.
21. Philipp Rümmer. Sequential, parallel, and quantified updates of first-order structures. In *LPAR*, pages 422–436, 2006.
22. Philipp Rümmer and Muhammad Ali Shah. Proving programs incorrect using a sequent calculus for java dynamic logic. In *TAP*, pages 41–60, 2007.
23. Jian Zhang and Hantao Zhang. Extending finite model searching with congruence closure computation. In *AISC*, pages 94–102, 2004.

An Experience Report on the Verification of Algorithms in the C++ Standard Library using Frama-C

Jochen Burghardt, Jens Gerlach, Hans Pohl, and Juan Soto

Fraunhofer FIRST, Kekuléstraße 7, 12489 Berlin, Germany,
FIRSTNAME.LASTNAME@first.fraunhofer.de,
URL: http://www.first.fraunhofer.de/device_soft_en

Abstract. Over the past few years, we have been conducting assessment studies to determine the utility of the Frama-C/Jessie platform of software analyzers (in conjunction with automatic theorem provers) for the formal verification of software. In this experience report, we discuss experiments in the verification of algorithms in the C++ Standard Library based on tool-supported Hoare-style weakest precondition computations to formally prove ACSL (ANSI/ISO C Specification Language) properties. Often automated provers are unable to perform inductive proofs. Hence, we introduce an approach to guide automated provers to find an inductive proof using auxiliary C-code corresponding to the proof structure. We also present a method to verify that a function only permutes the contents of an array, and obtain the relation between the pre- and post-index for each array element for use in later specification properties. Furthermore, we describe an approach to prove the essential properties of a function independent of each other, supplying for each task only the assumptions actually needed, i.e., related to the current goal. This approach reduces the proof search space and leads to higher verification rates for automatic provers. However, additional methods and tool support are desired to overcome drawbacks from a software engineering point of view. Finally, we sketch some ideas for an extension of ACSL for C++.

1 Introduction

As a step towards the goal of enabling verification of industrial software products, Fraunhofer FIRST is evaluating the Frama-C tool set within the Inter-Carnot-Fraunhofer project, DEVICE-SOFT[1]. Frama-C [1] is a suite of software tools dedicated to the analysis of C source code, developed at CEA LIST. Within Frama-C, the Jessie plug-in [2] enables deductive verification of C programs[2] that have been annotated with the ANSI-C Specification Language (ACSL) [9].

[1] Supported under grant 01SF0804 by BMBF (Germany) and ANR (France)

[2] Full ANSI-C is supported, including arbitrary aliases of type-compatible objects, but except for some current implementation restrictions concerning "dirty" type conversions, like `int <--> char*`, by casts or union types.

The paramount notion in ACSL is the function contract. While many software engineering experts advocate the "function contract mindset" when designing complex software, they generally leave the actual expression of the contract to run-time assertions, or to comments in the source code. ACSL is expressly designed for writing the kind of properties that make up a function contract.

The Jessie plug-in of Frama-C uses Hoare-style weakest precondition computations to formally prove ACSL properties of a program fragment. Internally, Jessie relies on the languages and tools contained in the Why platform [4]. Verification conditions are generated and submitted to external automatic theorem provers or interactive proof assistants. We employed the automatic theorem provers Alt-Ergo [13], CVC3 [8], Simplify [5], Yices [18], and Z3 [17] collectively.

We have chosen examples from the C++ Standard Library whose initial version was known as the Standard Template Library (STL). The STL contains a broad collection of generic algorithms that work not only on C-arrays but also on more elaborate containers, i.e., data structures. For this report, we will reference preselected algorithms, that were converted from C++ function templates to ISO-C functions that work on arrays of type `int`. Our experience report was inspired by our prior publishing, viz. the ACSL tutorial [12].

The structure of the remainder of this report are as follows. After a brief introduction to ACSL in Sect. 2, we demonstrate in Sect. 3 how to guide automatic provers to find difficult, in particular, inductive proofs. In Sect. 4, we report our experiences in proving permutations of array contents. Section 5 discusses our approach to prove certain properties of a function individually. In Sect. 6, we sketch some requirements for an object-oriented extension to ACSL, based on our experiences with downcasting C++ Standard-Library code to strict C. Finally, we draw some conclusions in Sect. 7.

2 An Introduction to ACSL

ACSL annotations are expressed in special C-comments `/*@...*/` as a multi-line comment or `//@...` as a single-line comment. A function contract declares a set of **`requires`** clauses, stating the properties the function may expect on entry, and a set of **`ensures`** clauses, stating the properties the function must satisfy upon exit (cf. Sect. 4 for examples).

Properties are formulas denoted in a language close to `C` itself. For example, equality, negation, and conjunction are denoted by `==`, `!`, and `&&`, respectively; binding-priorities are as in `C`. In addition, the weaker-binding junctors `==>` and `<==>` denote implication and equivalence; quantifiers over `C`- or logical types are denoted by `\forall TYPENAME VARNAME; FORMULA`, and similar for `\exists`. Moreover, relation chains familiar from mathematical notation, such as, `0 <= i < n`, may be used.

Function parameters and visible variables may appear in formulas, they refer, by default, to their values on entry and exit in a **`requires`** and **`ensures`** clause, respectively. The notation `\at(v,L)` refers to the value of `v` at the program point corresponding to the `C`-label `L:`. A predefined label `Old` allows one to refer

to on-entry values in **ensures** clauses too, `\old(EXPR)` being an abbreviation for `\at(EXPR,Old)`. The label `Here` refers to the on-exit value in an **ensures** clause.

In the function body, an **assert** clause may be placed at any program point, causing a corresponding additional proof obligation to be generated. Each loop may have a set of **loop invariant** clauses and a **loop variant** expression needed to prove its termination. The former property is expected to hold before loop entry and after the incrementation statement (usually "`i++`") of each loop iteration, while the latter expression must decrease at each iteration, but remain positive.

Using the default setting, `C`-types like `int` and `double` denote the finite ranges of values implemented on the target machine; absence of overflows is verified. In contrast, logical types like **integer** and **real** denote infinite sets like $\mathbb{Z}$ and $\mathbb{R}$ familiar from mathematics.

Programs may be enhanced with interspersed **ghost** declarations and statements that may compute auxiliary values used only for verification purposes. Since such **ghost** code is enclosed in the special comments, it is ignored by an ordinary compiler, like all ACSL constructs. Syntactical restrictions ensure that ghost code cannot influence non-ghost program components.

Auxiliary properties may be formulated as **lemma**s; their validity is checked by the provers. A macro-like mechanism, the **predicate** definition allows users to abbreviate arbitrary formulas by a parametrized name (Fig. 2). Parameters may be of `C` types or logical types enclosed in parentheses ( ). They may also denote memory states at certain labels enclosed in curly braces { }. If a definition or a lemma is declared to have just one such memory-state parameter, it may be omitted in the body; e.g. `bi[i]` in line 2 of Fig. 2 defaults to `\at(bi[i],L)`. The predefined predicate `\valid_range(a,l,u)` expresses the property that the addresses `&a[l]`, ..., `&a[u]` may be safely dereferenced at run-time.

3 Proof Assistance for Inductive Proofs by Automatic Provers

While Frama-C/Why supports interactive theorem provers like Coq, PVS, Isabelle/HOL, and Mizar, as well, we consider the additional necessity of learning their respective proving methodology and interactive command language a serious obstacle preventing an application-domain engineer from using such an approach. For this reason, we are exploring how far we can go using solely fully automatic provers, thus restricting the learning necessities to Hoare's verification method and the ACSL specification language.

We use an example from the heap operations in the C++ Standard Library to illustrate our discussion. Heap operations will also provide other examples in later sections.

The Apache C++ Standard Library User's Guide provides the following definition for heaps:

> A heap is a binary tree in which every node is larger than the values associated with either child. A heap and a binary tree, for that matter, can be very efficiently stored in a vector c, by placing the children of node i at positions $2i + 1$ and $2i + 2$. Using this encoding, the largest value in the heap is always located in the initial position, ...

The main operations include:

- `push_heap`, which inserts a new element into a heap;
- `pop_heap`, which removes the largest element from a heap;
- `make_heap`, which re-arranges an arbitrary array into a heap;
- `sort_heap`, which rearranges the elements in a heap so that they are in ascending order; and
- `heap_sort`, which uses `make_heap` and `sort_heap` to sort an arbitrary array in ascending order in time $\mathcal{O}(n \cdot \log(n))$. Note that this algorithm does not directly belong to the C++ Standard Library.

For detailed informal specifications of the aforementioned functions, cf. [6].

We explain our method along the example of the `pop_heap` function, which removes the root element `c[0]` of a heap, and reorders the remaining elements into a heap again. An essential property of `pop_heap` is that the popped value `c[0]` is in fact the largest one in the given heap.

Translating the informal specification of `pop_heap` to ACSL, we get

```
\forall integer i; 0 <= i < n ==>
        c[i] >= c[2*i+1] && c[i] >= c[2*i+2]
```

as a heap data-type invariant that will be abbreviated by a user-defined predicate `IsHeap(c,n)` in the following, and

```
\forall integer i; 0 <= i < n ==> c[0] >= c[i]
```

as "largest-value-in-initial-position" property.

A closer look reveals that an induction on i is necessary to prove the latter property from the former invariant. While that property is necessary to verify `heap_sort` in Sect. 4, none of the automatic provers employed by Frama-C is prepared to do induction proofs. Luckily, we found a way to trick them into it, utilizing Hoare's loop rule for that purpose. We also employed auxiliary (**`ghost`**) code and data. This situation is similar to many mathematical proofs, where auxiliary definitions are common.

We define a C-function `pop_heap_induction` that does not contribute to the functionality of `pop_heap`, but rather encapsulates the induction proof needed for the verification of that property. The `pop_heap_induction` function contract essentially requires our type invariant, viz. `IsHeap(c,n)` and ensures our proof goal, viz. the above "largest-value-in-initial-position" property:

```
/*@ ...
    requires IsHeap(c, n);
    ensures  \forall integer i; 0 <= i < n ==> c[0] >= c[i];
*/
```

```
void pop_heap_induction(const int* c, int n) {
    /*@ loop   variant n - i;
        loop invariant 0 <= i <= n;
        loop invariant \forall integer j;
                              0 <= j < i <= n ==> c[0] >= c[j];
    */
    for (int i = 1; i < n; i++) {
        //@ assert 0 < i ==> ParentChild((i-1)/2, i);
    }
}
```

In order to cause the prover to do an induction on `n`, we use a for-loop with a corresponding range and the induction hypothesis as loop invariant. The loop body is empty, except for an additional hint to the prover (viz. that `(i-1)/2` is the parent node of node `i`, employing another user-defined predicate).

In the function body of `pop_heap`, we call `pop_heap_induction` using a ghost statement, thereby establishing its inductive conclusion just at the place where it is needed for proof reasoning; cf. Fig. 1.

```
/*@
   requires 0 < n < (MAX_INT-2)/2;
   requires \valid_range(c, 0, n-1);
   requires IsHeap(c, n);
   ensures  IsHeap(c, n-1);
   ensures  c[n-1] == \old(c[0]);
   ensures  \forall integer i; 0 <= i < n ==> c[n-1] >= c[i];
*/
void pop_heap(int* c, int n) {
  //@ ghost pop_heap_induction(c, n);
  //@ assert \forall integer i; 0 < i < n ==> c[0] >= c[i];
  int max  = c[0];
  // ... (reordering loop omitted) ...
  c[n-1]  = max;
}
```

Fig. 1. C-function call to establish inductive consequence

Some care is necessary with the method described above. The set of provable consequences is not closed with respect to the deduction theorem, "Whenever $P \vdash Q$, then $\vdash P \to Q$", as long as there is not a Hoare rule like:

$$\frac{\{P\}\ \mathtt{S}\ \{Q\}}{\{true\}\ ;\ \{P \to Q\}} \qquad \text{if } \mathtt{S} \text{ doesn't change visible memory state.}$$

Note that `pop_heap_induction` changes the value of its local variable `i`. Therefore, a useful version of such a new rule needs to address the matter of visibility.

Due to the absence of such a rule, the relativized property

```
(\forall integer i; 0 <= i < n ==> c[(i-1)/2] >= c[i]) ==>
(\forall integer i; 0 < i < n ==> c[0] >= c[i]);
```

cannot be obtained by calling `pop_heap_induction`. However, we can establish it, if needed for further reasoning, by a modified version of `pop_heap_induction`, using this property as a post-condition and a similarly relativized loop invariant. In general, however, an additional assumption P may well lead some provers into an endless loop that were previously able to prove Q without it.

Our method is in principle not limited to the use of simple for-loops. For example, the following verification needs two nested loops:

```
/*@ ...
  requires \forall integer i; 0 <= i   < n-1 ==> c[i] <= c[i+1]
  ensures \forall integer i,j; 0 <= i <= j < n ==> c[i] <= c[j]
*/
void induction_example_2(const int* c, int n);
```

4 Permutations of Array Contents

This section is concerned with a method for proving that some procedure changes only the ordering of an array under consideration. For example, the function `pop_heap` should not just re-establish the heap property, but also ensure that all elements (except the one that was popped) remain in the heap, although possibly rearranged.

Our method relies on bijections (represented as index arrays) operating on the index set $\{0, \ldots, n-1\}$ for some n. Basically, we use a swap operation `swap(a, i, j)`, which simply exchanges elements `a[i]` and `a[j]` in an array, `a`. The corresponding predicate definition is as follows:

```
/*@ predicate Swap{L1,L2}(int* a, integer i, integer j) =
   0 <= i && 0 <= j
   && (\forall integer k; 0 <= k && k != i && k != j ==>
       \at(a[k],L1) == \at(a[k],L2))
   && \at(a[i],L1) == \at(a[j],L2)
   && \at(a[j],L1) == \at(a[i],L2);
*/
```

We now introduce some basic predicates and a lemma needed for our method as shown in Fig. 2. The predicate `Bijection` defines in the mathematical sense a bijection or one-to-one mapping from the set of natural numbers $\{0, 1, 2, \ldots, n-1\}$ to itself. In its current form it is suitable for the finite case only, therefore an upper bound `n` is needed. Lemma `B1` states that a bijection concatenated with a swap-mapping is still a bijection. The predicate `SameElements` is of the utmost

```
/*@ predicate Bijection{L}(int* bi, integer n) =
      (\forall integer i; 0 <= i < n ==> 0 <= bi[i] < n)
     && (\forall integer i, j;
       0 <= i < n && 0 <= j < n && i != j ==> bi[i] != bi[j]);

    lemma B1{L1,L2}: \forall int* bi,integer i, j, n;
      0 <= i < n && 0 <= j < n &&
      Bijection{L1}(bi, n) && Swap{L1,L2}(bi, i, j) ==>
      Bijection{L2}(bi, n);

    predicate SameElements{L}(int* a,int* o,int* bi,integer n)=
      Bijection{L}(bi, n) &&
      \forall integer k; 0 <= k < n ==> a[k] == o[bi[k]];
*/
```

Fig. 2. The bijection predicates and lemmata

importance, stating that the bijection `bi` shows how to reorder the indices to get the same values.

As examples, we use the functions `push_heap`, `make_heap`, and `heap_sort`, each showing different techniques. This approach works similar for `pop_heap` and `sort_heap`. We introduce four ghost declarations (see below).

```
//@  ghost int N;
//@  ghost int* biject;
//@  ghost int* twin;
//@  ghost const int* orig;
```

The global variable `N` plays a role later in `make_heap` (and `sort_heap` as well). For the moment, let us assume that `N` equals the actual size of the heap, `n`. The array `twin` behaves like a duplicate of the array `c` under consideration, see Fig. 3. Prior to the procedure, `c` and `twin` must have the same values at the same positions. The same must be true after the procedure. The elements in array `orig` can be thought of as being the elements of the original heap-contents `c`; they are retained and unaffected by any function verified here. The array `biject` will hold the bijection we seek. It is used to show that the `SameElements`-predicate with respect to `twin` and `orig` is preserved.

Figure 3 illustrates the principal operating scheme of the bijection method. It is based on an invariant property in the pre- and post-condition of all algorithms. The verification of this commonly occurring invariant is treated here.

Figure 4 shows our "universal function contract", in which is just a formalization of the relations shown[3] in Fig. 3. It is used literally in the same form for all functions involving heap-modifying operations. The symmetry between

[3] We omitted additional requirements that serve only to avoid numeric overflow.

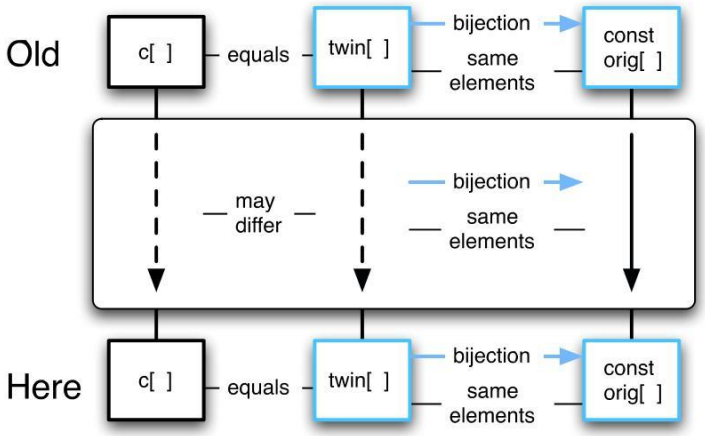


Fig. 3. Principal operating scheme of the bijection method

```
/*@ requires 1 <= n <= N;
    // \valid_range requirements ...

    requires \forall integer i; 0 <= i < N ==> c[i] == twin[i];
    requires SameElements(twin, orig, biject, N);

    ensures  \forall integer i; 0 <= i < N ==> c[i] == twin[i];
    ensures  SameElements(twin, orig, biject, N);
*/
```

Fig. 4. Contract for `push_heap`, `pop_heap`, `make_heap`, `sort_heap`, and `heap_sort`

requires and **ensures** reflects the vertical symmetry in Fig. 3. We distinguish the actual heap size, `n`, and an arbitrarily larger one, `N`. Since we require all operations to affect the area up to `N` this more relaxed requirement is easier to handle.

We start with the verification of `push_heap` whose implementation is shown below. This algorithm inserts the value found immediately beyond the old heap, viz. at `c[n-1]` into it.

```
void push_heap(int* c, int n) {
  const int tmp = c[n-1];
  int hole = n-1;
  /*@ loop invariant 0 <= hole < n;
      loop invariant tmp == twin[hole];
      loop invariant \forall integer i;
         0 <= i < N && i != hole ==> c[i] == twin[i];
      loop invariant SameElements(twin, orig, biject, N);
      loop   variant hole;
  */
  while (hole > 0) {
    const int parent = (hole-1)/2;
    if (c[parent] < tmp) {
      c[hole] = c[parent];
      //@ ghost swap(twin, hole, parent);
      //@ ghost swap(biject, hole, parent);
    } else
      break;
    hole = parent;
  }
  c[hole] = tmp;
}
```

The main part of its loop invariant corresponds to the **ensures** clauses of the universal contract, except for `i == hole`. We wish to be certain that `twin` is a duplicate of `c`. While `c` is manipulated with particular emphasis on run-time efficiency, we duplicate these manipulations in `twin`, however, using less efficient ghost calls to `swap` only. As we build-up `biject`, we apply swap-operators for both `biject` and `twin`. These two ghost assignments are the heart of the method.

Our implementation of `make_heap` is depicted below.

```
void make_heap(int* c, int n) {
  /*@ loop invariant 2 <= i <= n+1;
      loop invariant SameElements(twin, orig, biject, N);
      loop invariant \forall integer j; 0 <= j < N ==>
                     c[j] == twin[j];
      loop   variant n - i;
  */
  for (int i = 2; i <= n; i++)
    push_heap(c, i);
}
```

With successive calls to `push_heap`, elements are inserted into a heap `c` one by one. The essential **`requires`** and **`ensures`** clauses of the universal contract are just passed-on to its loop invariant. At this point we need to distinguish between `n` and `N`. Without the unique `N`, we would have predicates `SameElements(twin, orig, biject, i)` for different values of `i` which would not fit together.

The universal contract can also be used for `heap_sort`. This contract is most useful as a unique interface description in general, since it allows us to compose different algorithms easily. Since `heap_sort` is the outermost function in our example, we prefer, however, another version that is closer to the intended meaning of its informal description, as shown below.

```
/*@ requires 0 < n == N;
    // \valid_range requirements ...
    ensures  Bijection(biject, n);
    ensures  \forall integer i; 0 <= i < n ==>
        \at(c[i], Here) == \at(c[\at(biject[i], Here)], Old);
*/
void heap_sort(int* c, int n);
```

This alternative contract of `heap_sort` no longer requires `c` to equal `twin` initially nor `twin` to be a permutation of `orig`. Rather, we include ghost code in the implementation (below) to establish the following properties. First, we make both `twin` and `orig` a copy of `c`, and then we initialize `biject` to the identity mapping, viz. `[0,...,n-1]`. For the latter purpose, we call the `C++` Standard Library function `iota(a,n,v)` that assigns `v+i` to `a[i]` for $i = 0, \ldots, n-1$. Here, we require `n == N`, because the called functions deal with arrays of the same size.

```
void heap_sort(int* c, int n) {
   //@ ghost copy(c, n, twin);
   //@ ghost copy(c, n, orig);
   //@ ghost iota(biject, n, 0);
   make_heap(c, n);
   sort_heap(c, n);
}
```

As early as 1971, Hoare [16] was concerned about proving a rearrangement-only property of an algorithm. He suggested to introduce a concept of permutation, and to prove essential properties that might be re-expressed in ACSL as follows[4]:

[4] If they were used as a *definition* of `Permutation`, due to its recursivity its appropriateness relied on the implicit assumption that its intended semantics is the least fixpoint. Note that e.g. the greatest fixpoint was a predicate that is true for all `a` and `n`, which was certainly inappropriate.

```
\forall int* a, int n; Permutation{L1,L1}(a, n);
\forall int* a, int n, i, j; 0 <= i < n && 0 <= j < n &&
    Permutation{L1,L2}(a, n) && Swap{L2,L3}(a, i, j) ==>
    Permutation{L1,L3}(a, n);
```

If `Permutation{Old,Here}(a, n)` has been verified, the prover has just confirmed what could be seen also by code inspection, viz. that only `Swap`s were applied to `a`. Based on this property only, it seems to be impossible to prove properties of the rearranged array needed in a calling function. Let us assume that in our example an additional data-type invariant has to be proved, e.g., that every heap element has a value less than 10. This is straight-forward as soon as we are ensured that some additional array `biject` holds the source position each heap element came from[5]:

```
/*@ ...
    requires \forall integer i; 0 <= i < n ==> c[i] < 10;
*/
void foo(int* c, int n) {
   heap_sort(c, n);
   //@ assert \forall integer i; 0 <= i < n ==> c[i] < 10;
   // ...
}
```

Bubel et. al. [10] uses a different approach to characterize permutations for verifying selection-sort. They specify that the number of occurrences of each value remains unchanged in the rearranged array. This way, is it also possible to prove properties of the rearranged array needed in a calling function. In our example, no values larger than 9 may occur in the sorted array, if none occurred in the unsorted one.

5 Proving Properties Separately

As mentioned in Sect. 3, our experience corroborated again the well-known phenomenon that additional assumptions may lead a prover into an endless loop, even if they are not necessary for a proof. This is the first reason for separating function properties and their verification. The second reason follows a rationale of "separation of concerns" adapted from classical software engineering.

As an example, the function contract, `pop_heap`, essentially states that

1. the heap-property (each node entry is less or equal than its parent's entry) is re-established, and
2. all elements, except the one that was popped, remain in the heap.

[5] If the inverse bijection is needed in the verification of a calling function, it can also be constructed and validated, too.

We built two separate files, each containing a partial contract based on 1. and 2., and that share the same implementation code each, however with different loop invariants and other ACSL assertions, and varying auxiliary ghost code. While we were able to verify each of them successfully, an attempt to verify the whole contract (based on 1. and 2.) failed. We suppose that the reason was infinite applicability of some assumption from 1. to a proof goal from 2., or vice versa; however, certainty about that could only be gained by a detailed inspection of a prover trace, which is not available to us.

We consider our two partial contract proofs sufficient to convince a human user that the code has both properties 1. and 2. Moreover, each part is easier to understand on its own.

However, if `pop_heap` is to be used by another function `f` that needs `pop_heap`'s complete contract for its own verification proof, there is currently no sound way to verify `f`'s correctness. We can only declare (rather than define) `pop_heap` and claim (rather than prove) its entire contract in a preamble of the file containing `f`'s contract and code. This approach is unsatisfactory as soon as large software is to be verified or when a certifying authority is to be convinced of the correctness. Moreover, the need to maintain several copies of the implementation code is a serious drawback.

In order to exploit the advantages of separation and avoid its disadvantages, we suggest to supply explicit methodical support for users. Moreover, a tool should be provided that validates or establishes the syntactical restrictions and provides the entire contract for further use by the provers.

We thought of the following as a typical scenario: The tool maintains a "repository file" containing the whole contract and implementation, e.g. of `pop_heap`. In this file, ACSL clauses and ghost code lines may be annotated to indicate which of the partial contracts 1. and 2. they belong to. On user-demand, the tool extracts a particular "version", containing only the parts belonging to the selected partial contract, the proper source code, and the corresponding annotations. This version can then be updated stand-alone and finally be "checked in" using the tool, which tests consistency and reports any conflicts. Calling functions may rely on the complete specification.

6 Towards a Specification Language for C++

The C algorithms we have considered in our research have their origin in the C++ standard library. It would be very desirable to have a specification language for C++ with support for templates such that the *generic* algorithms and containers of the C++ standard library could deductively verified.

Based on our experience with ACSL we mention a few points that should be taken into account when defining a behavioral specification language for C++.

- Even if we stay within the realm of C programming it would make sense to extend ACSL to allow for *generic* predicates. Note that the built-in predicate `is_valid_range` is already generic with respect to the type of the values of an array.

- Initially, a specification language for C++ should use special comments to annotate source code. If later the specification language and the supporting tools are sufficiently mature, efforts could be undertaken to fully integrate formal specification and deductive verification into the language. This work could built on previous attempts to add "contract programming to C++" [14] as well as on the experiences with Spec# [3].
- An interesting aspect of the C++ standard library is that the informal specification includes requirements for the (amortised) complexity of algorithms. A specification language for C++ should include provisions to formally express the complexity of operations.
- The idea of using *concepts* to specify requirements for types had been introduced to C++ with the original STL. Despite strong efforts, concepts were excluded from the forthcoming C++ standard. A specification language for C++ should investigate whether *concepts* can contribute to concise and more general specifications.

Last but not least, we suggest that a specification language for C++ treats ranges in a way that fits more natural to the C/C++ language family. By this we mean the problem that in ACSL the set of valid indices of an array on length n, for example, must be specified as $0 \leq i \leq n-1$ whereas the established C-idiom describes this set as $0 \leq i < n$.

7 Conclusions

The deductive verification of the heap algorithms of the C++ standard library poses several challenges for automatic theorem provers. On the one hand, there is a need to perform *inductive proofs*. On the other hand, it must be shown that the heap operations only permute the elements in a range. For both problems we have found viable solutions that work well with the ACSL specification language and the Frama-C/Jessie tools.

Based on our experience with the deductive verification of C++ standard library algorithms we have also suggested tool-support for tackling larger verification tasks. The key idea here is the separate verification of independent properties.

It would be interesting to try to reproduce our methods in JML [7, 15, 11] and Spec# [3]. This can be expected both to yield an estimation of the generalizibility of our ideas and to compare the strengths and weaknesses of these different verification systems.

It may be a problem that the methods presented here are still too complex to be taught to application-domain engineers. However, in our opinion, they are still easier to learn than a language of proof tactics requiring much theoretical knowledge about an underlying logical calculus.

The idea of separation of concerns is widely accepted in software-engineering, and its extension to proof goals (Sect. 5) is expected to get so, too. The connection in Sect. 4 between manipulating the ghost array `twin` and its non-ghost

correlate c in an easy-to-verify and an equivalent fast-to-execute way, respectively, is familiar to programmers under the notion of "tweaking code for optimal performance". Obviously, the feature of ghost code is needed to implement the less efficient code variant.

References

[1] Frama-C Software Analyzers. http://frama-c.com
[2] Jessie Plug-in. http://frama-c.com/jessie.html
[3] Spec#. http://research.microsoft.com/en-us/projects/specsharp
[4] Why – Software Verification Platform. http://why.lri.fr
[5] Homepage of the Simplify Theorem Prover. http://freshmeat.net/projects/simplifyprover/ (2007)
[6] Standard Template Library Programmer's Guide. http://www.sgi.com/tech/stl (2010)
[7] Arnold, K., Gosling, J., Holmes, D.: The Java Programming Language. The Java Series, Addison-Wesley, Reading/MA (2000)
[8] Barrett, C., Tinelli, C.: Homepage of CVC3. http://www.cs.nyu.edu/acsys/cvc3/ (2010)
[9] Baudin, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ANSI/ISO C Specification Language, Version 1.4 Frama-C Beryllium implementation. http://frama-c.com/download/acsl-implementation-Beryllium-20090902.pdf (Sep 2009)
[10] Bubel, R., Hähnle, R., Schmitt, P.H.: Specification predicates with explicit dependency information. In: Beckert, B. (ed.) Proc. 5th Int. Verification Workshop (VERIFY'08). CEUR Workshop Proceedings, vol. 372, pp. 28–43. CEUR-WS.org (2008)
[11] Burdy, L., Cheon, Y., Cok, D., Ernst, M., Kiniry, J., Leavens, G.T., Rustan, K., Leino, M., Poll, E.: An overview of JML tools and applications. Technical Report TR NIII-R0309, Dept. of Computer Science, University of Nijmegen (2003)
[12] Burghardt, J., Gerlach, J., Hartig, K., Pohl, H., Soto, J.: ACSL by example. Tech. Rep. Version 4.2.1, Fraunhofer FIRST (Apr 2010), http://www.first.fraunhofer.de/owx_download/acsl-by-example-4_2_1.pdf
[13] Conchon, S., Contejean, E., Kanig, J.: Homepage of the Alt-Ergo Theorem Prover. http://alt-ergo.lri.fr/
[14] Crowl, L., Ottosen, T.: Proposal to add Contract Programming to C++. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1866.html (2005)
[15] Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java Language Specification. The Java Series, Addison-Wesley, Reading/MA (2000)
[16] Hoare, C.: Proof of a program: FIND. CACM 14(1), 39–45 (Jan 1971)
[17] Research, M.: Homepage of the Z3 SMT Solver. http://research.microsoft.com/en-us/um/redmond/projects/z3/
[18] SRI International: Homepage of the Yices SMT Solver. http://yices.csl.sri.com/

Formal Verification of Industrial C Code using Frama-C: a Case Study[1]

D. Pariente, E. Ledinot

Dassault Aviation - 78, quai Marcel Dassault - F-92552 Saint-Cloud Cedex 300, France
{Dillon.Pariente ; Emmanuel.Ledinot}@Dassault-Aviation.com

Abstract. This paper gives some results and lessons learnt with Frama-C, a static analysis toolbox, used to prove behavioral and safety properties on an industrial code. After a short presentation of the methods and tools background, the related industrial use case is briefly exposed, with an overview of the process that was followed. Then the positive results obtained so far are presented, with a few practices and additional tools developed in-house. To conclude, this paper presents some needs and future work directions that should be addressed, to ensure a technology readiness level compliant with operational use of formal verification into an industrial development environment.

Keywords: Software verification, Formal Methods, Hoare Logic, Abstract Interpretation, Theorem Proving.

1 Introduction

Since 1990, Dassault Aviation has carried out numerous formal methods studies and assessments. The first ones were focused on synchronous languages (first Esterel [1], then Lustre), for control and data flow formal specification, coding and model-checking, through collaborations with research teams. Over the last few years, much effort was devoted to the integration of UML modeling and signal flow programming (Matlab, Scade, Esterel), in order to introduce these new methods and tools into the Flight Control System (FCS) software development process. By the end of 2003, the first control module formally specified in a graphical way, automatically generated (~15 Kloc), and proven was embedded into a military aircraft operational software. More recently, in 2007, the FCS of the first Dassault's Fly-By-Wire business jet was developed and certified, using a similar development process.

In the meantime, some experiments on formal verification of hand-written code were initiated, because in numerous situations pieces of critical software cannot be generated from formal specification models (drivers, schedulers, encoding of data formats, ...). Indeed, even formally specified and automatically generated codes may take advantage of formal methods in many cases. For instance, as floating-point variables in data-flow models strongly compromise model-checking computability,

[1] *This work was partly supported by the French national Research project ANR/U3CAT 2008-SEGI-021-06, and the project DGAC/ANASTASY 2009-93-0816.*

these generated codes are good candidates for static analyses. This is the basis of our motivation to assess formal approaches, and to verify annotated hand-coded or automatically generated C programs (a first attempt is detailed in [2]).

The tools involved into the experimentation presented here are essentially developed through research projects RNTL/CAT and ANR/U3CAT, namely Frama-C toolbox [3], a recent but efficient collaborating static analysis platform. These tools are mainly developed by academics involved into these research projects (CEA LIST and INRIA ProVal).

In this case study, Frama-C has been used to prove the correctness of some properties annotated into a critical C code embedded into aircraft. This use case is representative of a certain class of programs and properties. Code contains both generated and hand-coded functions. Most of the sought properties are locally annotated to any function of the callgraph, but need a whole application analysis (i.e., they are context-sensitive to the main entry point), which is of course a challenging issue compared to unit proof (which generally only needs local function behavior to be discharged).

In the following, we will briefly introduce the underlying technical background and tools we have experienced for the last three years: Value Analysis, Jessie and Slicing Frama-C's plug-ins [3] using a common specification language named ACSL [23] inspired by JML, Why platform [7] (a verification condition generator developed at INRIA ProVal), and several automatic theorem provers like INRIA's Alt-ergo [8]. Then we will present our industrial use case that was successfully verified, and some techniques and "tricks" that permitted to overcome a few classical difficulties faced with formal methods implementations. To conclude, this paper will focus on lessons learnt and future work directions aiming Frama-C's usage into an industrial development context.

Of course, due to code development legacy and current industrial practices, the use case presented here is based on a C program. However, this experimentation is expected to be profitable in the context of other programming languages and verification tools, as theoretical issues, technical limitations, but also successful results obtained with C code, share large common features with other specification and coding languages (e.g., Spec# [4] for C#, JML [6] for ESC/Java2 [5], etc.), coming for sure with a bulk of new research challenges.

2 Background and Tools

This paragraph aims at giving some major references to theoretical background and tools used in our case study:

- the Frama-C platform [3] is an open source collaborative and extendable static analysis toolbox, coming with several plug-ins exploiting the same annotation language ACSL (ANSI-C Specification Language [23], inspired by JML), and implementing abstract interpretation method [9] (Value Analysis plug-in), deductive verification [10] [11] (Jessie plug-in), slicing [12], among other cooperating plug-ins,

– the Why [7] toolchain, a verification condition generator, interfaced with several automatic theorem provers like Alt-ergo [8], Z3 [13], Simplify [14], CVC3 [15], proof assistants like Coq [21], and some others.

Contrarily to many other analysis tools, Frama-C gives powerful means to achieve proof of properties using different cooperating methods. Hence, users are less potentially facing implementation limitations of classical mono-paradigm-based static analysis tools, as they may switch to another method and plug-in when the currently used one is not conclusive w.r.t. the sought properties.

Frama-C also offers ways to develop user-specific plug-ins, to get around some costly hand-made operations by automating them, or to palliate a few sources of non-conclusive results, from the code annotation phase, to some customized static analyses themselves (these aspects will be discussed later).

2.1 Preamble on Static Analysis Methods

Static analysis [16] is the analysis of computer software, performed without actually executing programs, and generally by an automated tool. Uses of the information obtained from the analysis vary from highlighting possible coding errors, to formal methods that mathematically prove properties about a given program (e.g., code behavior matching its specification). A growing commercial use of static analysis is the verification of properties on software used in safety-critical computer systems, and locating potential vulnerability in code.

It has been proven that, except for some hypothesis that the state space of programs is finite and small, finding possible run-time errors or more generally any kind of violation of a specification on the final result of a program is undecidable: there is no mechanical method that can always answer whether a given program may or may not exhibit runtime errors (works of Church, Gödel and Turing in the 1930s). As with most undecidable questions, one can still attempt to give useful approximate solutions. These solutions can be achieved using different formal techniques like:

– Model-checking [17], usually devoted to finite or reduced state space,
– Abstract interpretation [9], generally based on a preliminary data-flow analysis, providing an over-approximation of program behaviors, simpler to analyze, sound (every property true of the abstract program can be mapped to a true property of the original program), but incomplete (not every property true of the original program is true in the abstract program),
– Deductive verification (by means of assertions in source code as first suggested by Floyd-Hoare logic), verification condition generation, and theorem proving,
– Slicing, consisting in keeping all the statements that affect a variable v at a given statement (v is a variable of the program).

2.2 Overview of implementations in Frama-C platform

Frama-C gathers several static analysis techniques (abstract interpretation, deductive verification, slicing, …) in a single collaborative framework. Namely, the cooperating

plug-ins approach of Frama-C allows static analyzers to make use of the results already computed by other analyzers in the same framework.

Abstract Interpretation.
The Value Analysis plug-in [3] computes for each variable, a set of values which necessarily contains the values obtained on any concrete execution. It is quite automatic, although the user may guide the analysis in places. It handles a wide spectrum of C constructs. This plug-in uses abstract interpretation techniques.

The results of the value analysis are accessible to the other plug-ins (including those developed by final users). Furthermore, in order to propagate its computations as far as possible, Value Analysis plug-in may generate its own annotations. These annotations deals with some potential runtime errors that must be refuted by means of any other plug-in (as deductive verification plug-in Jessie, for instance).

Deductive verification - Hoare logic.
Hoare logic [10] (a.k.a. Floyd–Hoare logic) is a formal system whose purpose is to provide a set of logical rules in order to reason about the correctness of computer programs with the rigor of mathematical logic. The central feature of Hoare logic is the well known Hoare triple {P} C {Q} where P and Q are assertions and C is a command. P is called the pre-condition and Q the post-condition: the triple is valid if in any state where the pre-condition holds, executing the command establishes the post-condition. Assertions are formulas expressed in predicate logic.

Weakest-Precondition (WP) [11] calculi allow to automate reasoning in Hoare logic: WP(C,Q) is a formula such that it suffices to prove P=>WP(C,Q) to guarantee validity of {P} C {Q}. P=>WP(C,Q) is a verification condition. Verification conditions (VCs) or proof obligations (POs) can be validated by theorem provers (automatic ones based on decision procedures, SMT, or proof assistants, like Coq). Jessie is a deductive verification plug-in developed in Frama-C, which supports a fairly large subset of C, including function calls, and arbitrary pointer aliasing thanks to a Burstall-Bornat [18] memory model and a separation analysis [19].

For instance, the rule for Weakest-Precondition (WP) computation in case of function call is defined as follows, with the - simplified - notations below (these notations will be re-used later on this paper):

- let R_f, A_f, and E_f be respectively the *Requires* (pre-condition), *Assigns* (effects) and *Ensures* (post-condition) clauses of a function f,
- let M_{Li} be a (simplified) memory state at control point L_i,
- let Assigns(M_{Lx}, M_{Ly}, A_f) be a predicate meaning: A_f is an over-approximation of modified memory locations, from M_{Lx} state to M_{Ly} state,
- X{@M_{Li}}: any predicate X whose parameters are evaluated at memory state M_{Li}.

With these notations, the WP at memory state M_{Li} for a property P upwarded through a call to a function f(...) is:

$$WP\{@M_{Li}\}(\ f(...)\ ,\ P\ ) =$$
$$R_f\{@M_{Li}\}\ /\backslash\ (Assigns(M_{Li},\ M_{Li+1},\ A_f)\ /\backslash\ E_f\{@M_{Li+1}\}\ =>\ P)$$

Once again, this is a simplified expression to ease the reading, as we got rid of some important memory modeling features, and especially related to separation logic (see [25] for more details).

Slicing.
As presented earlier, slicing is the computation of parts of a program that may affect the values computed at some point of interest, referred as a slicing criterion. In Frama-C, these criteria can be either a statement, or annotations, or a particular read/written variable at the return point of a given function, and so on. The Slicing plug-in produces an output program made of a subset of the statements of the analyzed program, in the same order. The output program is guaranteed to be compilable C code, and to have the same behavior as the analyzed program from the point of view of the provided slicing criterion.

3. Brief presentation of the use case and results obtained

In this paragraph, we will not focus on the industrial context of the case study, nor on a detailed description of the application that was analyzed, as it comprises confidential aspects that could not be outlined in this paper. Let us simply say that this use case is based on:

- a real critical embedded "control program" application, in the class of *n*x10 Kloc in C language, less than 200 C functions, mainly generated by Scade/KCG [20], with some low-level hand-coded functions,
- several hundreds of properties contributing to software robustness verification (for instance, ensuring that some input value domains fit function requirements, like verifying that SQRT function parameter is always positive whatever the calling context).

The main goals of this study are to assess (feasibility and) productivity gain due to automation of code analysis and property verification. The challenges are related to the size of the code under formal analysis, the number of properties to prove, the methodology to define w.r.t. the different Frama-C plug-ins to exploit (even though the soundness of plug-ins' cooperation is an academic task still in progress at this time). Furthermore, cutting in costs resulting from program changes, and helping users during code review, also appear as important features. As well, exhaustive value domain coverage performed by static analysis (intrinsic property of formal methods) is found to be an important advantage in comparison to classical program testing which is selective, non exhaustive, by definition.

Process and results.
The methodology applied to the aforementioned use case is quite straightforward, from a high level point of view. It consists in the following general steps:

– step A: Annotating the code with functional properties as assertions on calling contexts, or as pre-conditions of called functions,

- step B: Adding some "auxiliary" annotations (to improve the next "step C" accuracy defined below), if needed,
- step C: Processing the annotated code by abstract interpretation, using Frama-C's Value Analysis plug-in,
- step D: Discharging annotations generated by Value Analysis, and auxiliary ones, by means of Frama-C's Jessie plug-in, Why toolchain, and finally some automatic theorem provers.

Of course, in practice, these steps are not realized so simply. Typically, steps A and B need more experience when expressing properties to facilitate Value Analysis process, to improve result accuracy, to ease VC generation and to make decision procedures conclusive (sometimes by adding *ad hoc* lemmas). It generally requires several iterations before getting the expected results. Nevertheless, 100% of properties defined in this case study were formally proved valid, all the auxiliary annotations also discharged (using several automatic theorem provers to cover all the different VCs), and so, feasibility acquired. Note that some lemmas, especially dealing with non-linear arithmetic over reals and necessary to help automatic provers, were manually discharged or proved by means of Coq proof-assistant.

Several additional techniques and tools (some developed "in-house" at Dassault Aviation, as specific Frama-C's plug-ins) were necessary to achieve this result for our real-sized case study. These last are discussed in the following paragraph.

4. Some common difficulties and solutions to get around

We present below some of the techniques and *tricks*, developed and applied to overcome some limitations and usual difficulties faced when formally proving large source codes or more sophisticated properties.

4.1 Annotating automatically source codes

At first, we developed a plug-in which generates and inserts automatically some ACSL annotations into the C code. These annotations are somehow tied to our inner coding rules. This plug-in alleviates user's burdening annotation activity, but mandatory for deductive verification. GENA-annot (the name of this plug-in), for all C functions of the program, gives some:

- pre-conditions (*requires* clauses) related to pointer and array validities,
- function effects (*assigns* clauses),
- simple loop invariants and variants related to loop iterators.

So far, the amount of these needed annotations for our industrial code seems to correspond more or less to 1/3 of the program size. This should make obvious the interest of this automatic generation, as it avoids a fastidious and costly annotating task. This plug-in uses some of the results of Value Analysis plug-in in order to

propose more accurate clauses (in particular for loop invariants, by estimating iterator bounds). Its execution time is negligible as most of CPU time is indeed due to Value Analysis process.

For instance, the example code below presents a function `copy()`, with no annotation:

```
void copy(int *o,int *d)
{ int i, n=g(o,d);
  for(i=0; i<n; i++) { d[i] = o[i]; }
}
```

Let us assume that `n=g(o,d)` statement is interpreted by Value Analysis and yields `n in {5}`. This information, which determines loop bound as `i<n` in the loop condition, is directly exploited by GENA-annot. Thus, the same function, but now automatically annotated by GENA-annot (note that specifications in ACSL are special C comments starting with "`/*@`" or "`//@`") is as follows:

```
/*@ requires \valid_range(o,0,4)&&\valid_range(d,0,4);
    assigns d[0 .. 4]; */
void copy(int *o,int *d);

void copy(int *o,int *d)
{ int i, n=g(o,d); // Value Analysis returns n in {5}

/*@ loop invariant 0<=i<=5; loop assigns d[0 .. 4];
    loop variant 5-i; */
  for(i=0; i<n; i++) { d[i] = o[i]; }
}
```

The declaration of function `copy()` is put into a separated header file, to be accessible to other files needing `copy()`'s contract. As it can be easily seen in this example, not all of the annotations necessary to prove `copy()` correctness are of course generated. But the main *simple* ones are provided. It is up to the user to fill and complete the specification of this function. Other promising approaches for automatic annotation generation tied to Frama-C and ACSL language can be found in [25][31].

4.2 Validating lemmas in ACSL, by means of symbolic computation tool

It is a common practice to add some lemmas, written in ACSL and inserted into source code, to ease the task of ATPs when discharging proof obligations. These lemmas are often related to arithmetic properties, and will be exploited by SMT provers. The problem is to justify or even prove these lemmas to ensure their correctness. Such lemmas are not accessible to SMT solvers. Jessie plug-in then allows to call interactive provers such as Coq [21] to prove these lemmas. But this can be difficult and costly to do, as final users are not always familiar with proof assistant usage. To get around this point, we developed a plug-in, named GENA-lemmas, which translates ACSL lemmas (e.g., Fig. 1) into Maxima [22] hypotheses and goals, and automatically launches this symbolic computing tool on them (see Fig. 2). Note that for the moment, correctness of the translation from ACSL to Maxima is not guaranteed (and of course, it is not complete and fails if the lemma is too complicated

or does not fit with a subset of ACSL syntax): its purpose is only to give a first *friendly* indication on soundness.

```
/*@ lemma qr7_7a :
  \forall real u,yi,ys,xi,xs; (xi<=u<xs) && (yi<ys)
  ==> yi<=(((u-xs)*(yi-ys))/(xi-xs))+ys; */
```

Fig. 1. A lemma in ACSL.

```
assume(xi <= u);assume(u < xs);assume(yi < ys);
(yi <= ((u-xs)*(yi-ys))/(xi-xs)+ys), pred;
maybe(%);

/* Maxima answers:
(%i2,%i3,%i4) assume(xi<=u)&&(u<xs)&&(yi<ys)
                        (u - xs) (yi - ys)
(%i5) ev(yi <= ys + ------------------,pred)
                          xi - xs
(%o5)                       true */
```

Fig. 2. ACSL and Maxima lemmas (after translation). In comments, Maxima results.

4.3 Subdividing input interval for more precise output results

When dealing with non-linear arithmetic or in presence of multiple variable instances into the same arithmetic expression, Abstract Interpretation is often non-conclusive as it over-approximates too largely output intervals. To illustrate that point, interpreting a simple example like: `y = x-x;` with `x in [-1;1]`, a non-relational and/or not "customized" abstract interpreter will give: `y in [-2;2]`. There are several means to palliate this over-approximation. Either it is possible to add some *ad hoc* heuristics into the abstract interpreter, or one can add an assertion (simply "`//@ assert y == 0;`") to reduce the state of values (this is possible with Value Analysis plug-in) and then discharge this "auxiliary" assertion by other means. Or one can also subdivide the input interval and require Value Analysis to propagate states separately (w.r.t. control flow or any logic disjunction in an assertion).

This subdivision approach can, in certain cases, be done by hand. For instance, with the following code, and variable `x in [-10.0;10.0]`:

```
//@ assert -10.0<=x<=10.0;
y = x * x;
//@ assert y>=0.0;
```

the second assert is not proved valid with Value Analysis plug-in (`y`'s possible values are over-approximated in `[-100.0 ; 100.0]`). One has to modify the first annotation by introducing a disjunction of input intervals:

```
//@ assert -10.0<=x<=0.0 || 0.0<=x<=10.0;
y = x * x;
//@ assert y>=0.0;
```

Now, the second assertion (`y>=0.0`) is valid, because Value Analysis is able to propagate each term of the disjunction in the first assertion separately when evaluating the second assertion (i.e., before computing "union" of domains): `y`'s possible values are now in the interval `[0.0 ; 100.0]`.

With more arithmetically sophisticated expressions, these manual subdivisions can not be written by hand because these may be too numerous or complex to compute. To address this point, we developed in-house a plug-in named GENA-subdiv which automatically generates sub-intervals when the sought property is violated.

Let us take for instance a Taylor's series cosine implementation with self-explanatory ACSL annotations:

```
//@ requires -pi<=x<=pi;
float my_cosine(float x)
{ float y, x2 = x*x;
  float x4 = x2*x2;
  y = 1 - x2/2 + x4/24 - x4*x2/720;
  //@ assert -1.0<=y<=1.0;
  return y;
}
```

For this annotated code, GENA-subdiv generated 1600 intervals, necessary to prove the assertion with Value Analysis, in about 30 seconds (with a CPU at 2 GHz).

Indeed, the subdivision process is "lazy": the input interval for `x` is subdivided if and only if it invalidates the sought property. This means that if `x` is in `[a;b] U [b;c]`, and only `[a;b]` invalidates the assertion, then only `[a;b]` will be subdivided into several sub-intervals at the next subdivision iteration. This entails an important saving of CPU time and memory resources. Furthermore, GENA-subdiv contains heuristics and options that permit to specify symmetry or periodicity properties for the given C function, which also reduces drastically subdivision computation time (e.g., cosine being symmetrical w.r.t. y-axis, then only subdivisions of `[0;pi]` must be considered). Of course, GENA-subdiv also comes with stop conditions (maximum number of subdivision iterations, CPU time limit, ...) to guarantee termination of the process.

4.4 Introducing cut strategies in Jessie usage

In previous experiences [2] with CEA's CAVEAT tool [24], we experimented cut strategies whose goal is to alleviate WP computations, especially in case of large code analysis. In some cases, it is difficult to prove a sought property because of WP clause size growing rapidly when propagated upward in the source code. This is sometimes the case with functions generated from Scade models, in which a lot of connections between several nodes (i.e., functions) are naturally implemented by numerous assignment statements. This is also the case when using the Leino's quadratic WP computation algorithm [26] (usefully implemented into Why compiler to deal with source codes containing lots of if/else statements): the WP computation can provide provers with a smaller number of proof obligations (POs), but these POs are bigger in

size, even more with a lot of assignments (due to the need of numerous memory state variables).

Indeed, in our case study, many properties need not the entire code of the function to be discharged, but just some function calls or block contracts. To illustrate the purpose, in what follows, we assume that an assertion `P`, to be proved, just needs some of the statements of a function `f()`, let us say, only the contract related to the call to `f5()`:

```
          void f(...)
          {
L1:           f1(...);
L2:           f2(...);
L3:           f3(...);
L4:           f4(...);
L5:           f5(...);
L6:         //@ assert P;
          }
```

A first attempt should be to slice the source code of `f` according to assertion `P`, in order to get rid of calls to `f1()`, `f2()`, `f3()` and `f4()`. Frama-C's slicer generally gives very efficient results. But this does not always generate the expected "tiniest" code that would allow quick proof by provers: some statements are still present in the sliced code and may be responsible of huge WP computations. Another try is to use Why's hypothesis pruning algorithm [32]: it permits to reduce size and complexity of POs through context pruning w.r.t. variables and predicates involved into the goal of the sought POs. Certain classes of POs are good candidates for this kind of simplification. Unfortunately, some are still difficult to compute in a reasonable amount of time and memory resources.

With the same notations as in §2.3, to prove assertion P, we have to discharge the following WPs (formulas below are intentionally simplified regarding memory model and separation, see [25] for more details):

WP at control point L5:

$$WP\{@M_{L5}\}(\ f5(...)\ ,\ P\ ) = R_{f5}\{@M_{L5}\}\ /\backslash\ (Assigns(M_{L5},\ M_{L6},\ A_{f5})\ /\backslash\ E_{f5}\{@M_{L6}\}\ =>\ P)$$

WP at control point L4:

$$WP\{@M_{L4}\}(\ f4(...)\ ,\ WP\{@M_{L5}\}(\ f5(...)\ ,\ P\ )\ ) = R_{f4}\{@M_{L4}\}\ /\backslash\ (Assigns(M_{L4},\ M_{L5},\ A_{f4})\ /\backslash\ E_{f4}\{@M_{L5}\}\ =>\ WP\{@M_{L5}\}(\ f5(...)\ ,\ P\ ))$$

... and so on, until the entry point of function `f()`.

One could easily guess that these computations might become difficult to manage in size and complexity (by either VC generators or provers), and sometimes useless since $E_{f5}\{@M_{L6}\}$ => P could be indeed trivially discharged.

There is an obvious way to prove this kind of property, alleviating computing resources, which consists in adding some ACSL *statement contracts* [23] on function calls (or blocks of function calls) thus hiding the complexity of the annotated code.

Let us take again the example above, but now with new statement contracts on function calls which are not useful when discharging the POs related to assertion `P`:

```
          void f(...)
          {
L1:           /*@ ensures \true;*/ f1(...);
L2:           /*@ ensures \true;*/ f2(...);
L3:           /*@ ensures \true;*/ f3(...);
L4:           /*@ ensures \true;*/ f4(...);
L5:           f5(...);
L6:           //@ assert P;
          }
```

These annotations do not modify the behavior of function `f()` as they are only comments. Whatever the effects of the functions `f1()` to `f4()`, they are now considered as ensuring "true" and assigning any location (this is the default when mentioning no *assigns* clause in ACSL), and most of all they hide the corresponding function contracts (i.e., during WP computations for assert `P`, only the *ensures* clauses are taken into account: they permit to consider function calls at this step as *black-boxes*). This allows to simplify drastically the proof of assertion `P` as:

WP at control point L1:

$$WP\{@M_{L1}\}(\ \{all\ f's\ statements\}\ ,\ P\ ) = R_f\ /\backslash\ (Assigns(M_{L1}, M_{L6}, A_{f5})\ /\backslash\ E_{f5}\{@M_{L6}\}\ =>\ P)$$

The POs generated by the "ensures \true;" are obviously trivially discharged.

In our use case, we faced a function generated from a Scade model, with some assertions (that could be simply "locally" discharged as `P` in `f()`), with 20 calls to other functions, and lots of assignments before and after each function call. This function with its assertions, once analyzed by Jessie and Why, gave a huge number of POs (precisely: 19 456 obtained after automatic splitting of conjunctions in goal conclusions), from which an important part could not be discharged in a "acceptable" amount of time (and even made Simplify prover crashing with an unusual "subscript out of range" error message). Once the same original function code is annotated with ACSL statement contracts, we only obtain two POs discharged in less than 10 seconds by most provers.

In the same way, we developed another Frama-C plug-in which generates ACSL contracts for `if/else` statements in source code. This plug-in aims at reducing the complexity of WP computations, even in case of quadratic WP mentioned before. The short example below presents some *assigns* and *ensures* clauses generated (using Value Analysis results) automatically for any `if/else` statement:

```
// ...
  //@ assigns V10; ensures 0<=V10<=1; // generated by GENA
  if (f28->M375) { if (! f28->M282) { if (f28->M388) { V10 = 1;
} else V10 = 0; } else { V10 = 0; }  }
  else { V10 = 0; }
// ...
```

Fig. 3. Effects and post-condition generated automatically for an `if/else` statement.

Readers used to WP computations across conditionals will certainly grasp the interest in replacing some relevant `if/else` statements by contracts, and so *black-boxing* once again the C constructs known to introduce more complexity in WP.

4.5 Mixing Value Analysis, Jessie and *ad hoc* C functions devoted to proof

In our case study, some properties deal with large size arrays. The related verification conditions (VCs) generated are difficult and in some cases impossible to discharge by automatic provers (facing difficulties when handling inductive lemmas, etc.). For instance, proving that a big array given in extension is sorted may require huge computational resources, sometimes unaffordable! According to tools' developers, this is a specific issue that might be solved by improving the automatic provers.

Frama-C offers the ability to combine abstract interpretation with deductive verification to get around many issues revealed when attempting to prove VCs: one can interleave assertions that will be proved correct by either Value Analysis or Jessie plug-ins, alternatively. In other words, some assertions validated by Value Analysis will alleviate Jessie and automatic provers workload, and vice-versa. But in some cases, automatic provers are still unable to discharge the VCs.

Therefore, our approach consisted in defining additional C functions:

- they are interpreted by Value Analysis and referenced in program as ACSL ghost code (statements that are *understood* by analyzers but do not change original code behavior), in order to compute, say, some logic variable *V* domain,
- they have a specification *S* validated by Jessie, useful to ensure some auxiliary assertion *A*(*V*) (i.e., *A* makes reference to *V*), such as *A*(*V*)==>*P* (*P* being the sought property).

In our case study, we successfully applied this mechanism to prove some properties involving big-sized arrays. The principles can be illustrated in the following generic example:

```
// implementation of a C function devoted to proof
//@ ensures S;
float ghost_function(...) { ... }

// main function
void main( )
{
...
//@ ghost float V = ghost_function(...);
```

```
    // ghost_function and V are interpreted
    // by Value Analysis

//@ assert A(V); // validated by Jessie, as "S==>A(V)"

//@ assert P;    // validated by Value Analysis
                 // with the help of A(V)
}
```

All the lemmas, logic functions and predicates needed to prove this kind of annotated code, of course, had to be defined, and will not be presented here for convenience. This approach is expected to solve many common difficulties faced, not only by automatic provers, but also by final users when attempting to express complex properties only with logic. By using additional C functions, given the fact that abstract interpretation is able to analyze accurately their related code, it is the whole C language expressiveness that can be exploited to write and validate formally properties (in a sense, a similar approach is model-checking for which it is possible to define *observers* with the same syntax and semantics as the ones used for the model).

5. Conclusion

This paper has presented the process and results of a use case involving formal methods and Frama-C toolbox applied to a real critical code verification. This case study is representative of a class of programs and properties commonly developed and embedded into aircraft. To succeed in verifying the sought properties, some additional developments and methodologies were required, to address several issues often met in formal method implementations. It is worth mentioning that some hypotheses needed to be expressed, for instance on plug-ins' memory model compliancy, or on numerical accuracy aspects, which are indeed on-going works, and so, out of the scope of this paper for the time being.

Of course, for any other class of code or annotation, other tricks will have to be found, and surely new plug-ins will give a hand to users by simplifying preliminary or even intermediary static analysis workload. This is one of the strong benefits from Frama-C toolbox, as it allows users to develop their own plug-ins, in a multi-paradigm and collaborative context, based on ACSL, a common annotation language.

At this time, the whole tools are not yet fully industrialized, even if very well documented and supported by a researcher community, at least through discussion mailing-list of high reactivity. Indeed, these very recent tools still lack of integration w.r.t. results obtained by the different plug-ins. This is palliated for the moment by more methodological effort and sometimes a fastidious manual management of the results obtained. Hopefully, this will be made easier through a future Integrated Proof Environment development planed for the next months in the context of project ANR/U3CAT. From an industrial point of view, this work was done in a Research & Technology environment, with less critical resource constraints than in operations (like for instance no concrete integration into the development process). This issue is

addressed as well in on-going projects involving development teams at Dassault Aviation.

To improve software quality and safety, intensive testing had been unavoidable for years, even for unit function validation. Recent works, in particular done by Airbus with CEA's CAVEAT tool [29], opened some ways for formal and automatic verification usage. With new generation tools like Frama-C, formal method implementations can be even more firmly considered to replace testing activities in some identified cases. This is enforced by latest breakthroughs of static analysis in airworthiness certification of critical embedded software norm on-going specification (DO-178C norm project [27], to be released before the end of 2010): it is expected to benefit from certification credits, that is, replacing function testing by formal verification, conditioned of course by the qualification of related formal tools.

Acknowledgements
The authors are grateful to Benjamin Monate (CEA LIST) and Claude Marché (INRIA ProVal), and their groups, for their involvement in adapting research results to industrial needs, and their efficient technical support on Frama-C platform.

References

1. G. Berry, E. Ledinot & al., “Esterel: a formal method applied to avionic software development”, Sc. of Computer Progr., v.36 n.1, p.5-25, Jan.1.2000
2. E. Ledinot, D. Pariente, Formal verification of manual code: some industrial needs and recommendations, Embedded Real Time Software, SIA ed., 2006 (http://www.sia.fr/dyn/publications_detail.asp?codepublication=R-2006-01-3A3)
3. http://frama-c.com/index.html
4. http://research.microsoft.com/en-us/projects/specsharp/
5. http://kind.ucd.ie/products/opensource/ESCJava2/
6. http://www.eecs.ucf.edu/~leavens/JML/
7. J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In W. Damm and H. Hermanns ed., 19th Int. Conf. on Computer Aided Verification, LNCS, Berlin, Germany, July 2007. Springer-Verlag.
8. S. Conchon, E. Contejean, J. Kanig, and S. Lescuyer, {CC(X)}: Semantical Combination of Congruence Closure with Solvable Theories, Proceedings of the 5th International Workshop on Satisfiability Modulo Theories, SMT 2007, Electronic Notes in Computer Science, Elsevier Science Publishers, vol. 198-2, pp 51-69, 2008. (http://alt-ergo.lri.fr)
9. P. Cousot, R. Cousot, “Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints”, 4th ACM SIGACT-SIGPLAN POPL, p.238-252, January 17-19, 1977, Los Angeles, California.
10. C. A. R. Hoare, “An axiomatic basis for computer programming”, Volume 12 , Issue 10, p576 - 580, ACM Press New York, NY, USA October 1969.
11. Edsger W. Dijkstra, Guarded commands, nondeterminacy and formal derivation of program. Communications of the ACM, 18(8):453–457, August 1975.
12. Mark Harman and Robert Hierons. "An overview of program slicing", Software Focus, Volume 2, Issue 3, pages 85–92, January 2001.
13. L. de Moura and N. Bjørner. Z3, An Efficient SMT Solver. http://research.microsoft.com/ projects/z3

14. D. Detlefs, G. Nelson, and J. Saxe. Simplify theorem prover. http:// research.compaq.com/src/esc/simplify.html
15. Clark Barrett and Cesare Tinelli. CVC3. In Werner Damm and Holger Hermanns, editors, Proceedings of the 19th International Conference on Computer Aided Verification (CAV '07), volume 4590 of Lecture Notes in Computer Science, pages 298-302. Springer, July 2007. Berlin, Germany.
16. Nathaniel Ayewah, David Hovemeyer, J. David Morgenthaler, John Penix, William Pugh, "Using Static Analysis to Find Bugs," IEEE Software, vol. 25, no. 5, pp. 22-29, Sep./Oct. 2008, doi:10.1109/MS.2008.130.
17. Symbolic Model Checking, Kenneth L. McMillan, Kluwer, ISBN 0-7923-9380-5.
18. R. Bornat. Proving pointer programs in Hoare logic. In Mathematics of Program Construction, pages 102–126, 2000.
19. Thierry Hubert and Claude Marché. Separation analysis for deductive verification. In Heap Analysis and Verification (HAV'07), Braga, Portugal, March 2007. (http://www.lri.fr/~marche/hubert07hav.pdf)
20. http://www.esterel-technologies.com
21. Coq'Art: The Calculus of Inductive Constructions. Series: Texts in Theoretical Computer Science. An EATCS Series. Y. Bertot, P. Castéran. Springer, 2004. ISBN: 978-3-540-20854-9
22. http://maxima.sourceforge.net/documentation.html
23. P. Baudin, J.-C. Filliâtre, T. Hubert, C. Marché, B. Monate, Y. Moy, and V. Prevosto. ACSL : ANSI ISO/C Specification Language v1.4, 2010 (http://frama-c.com/acsl.html)
24. CAVEAT project. http://www-drt.cea.fr/Pages/List/lse/LSL/Caveat/index.html.
25. Y. Moy. PhD thesis. Automatic Modular Static Safety Checking for C Programs. Université Paris-Sud, 2009. (http://www.lri.fr/~marche/moy09phd.pdf)
26. K. Rustan M. Leino. Efficient weakest preconditions. Inf. Processing Letter, 93(6):281–288, 2005.
27. http://en.wikipedia.org/wiki/DO-178
28. Edmund Clarke & al., Predicate Abstraction of ANSI-C Programs Using SAT, Formal Methods in System Design, v.25 n.2-3, p.105-127, September-November 2004.
29. Applying Formal Proof Techniques to Avionics Software: A Pragmatic Approach, F. Randimbivololona & al., FM'99, Toulouse, September 1999.
30. C. Marché, The Krakatoa tool for Deductive Verification of Java Programs, Winter School on Object-Oriented Verification, Viinistu, Estonia, Jan. 2009. (http://krakatoa.lri.fr/ws)
31. Y. Moy and C. Marché, Modular Inference of Subprogram Contracts for Safety Checking, Journal of Symbolic Computation, 2010, to appear.
32. J.-F. Couchot and T. Hubert, A Graph-based Strategy for the Selection of Hypotheses, FTP 2007, International Workshop on First-Order Theorem Proving, Liverpool UK, Sep. 2007.

Verification of Variable Software: An Experience Report *

Richard Bubel, Crystal Din and Reiner Hähnle

Department of Computer Science and Engineering
Chalmers University of Technology
bubel@chalmers.se, crystal@student.chalmers.se, reiner@chalmers.se

Abstract. We report on our experiences with formal specification and verification of variable and customizable software realized in a software product family architecture using the Java Modeling Language (JML) and the KeY verification system. Software product families can be adapted to different deployment scenarios and provide instantiable feature sets as requested by the customer. Along a small case study we explore how to generate JML specifications for/from a given feature configuration and report on verification attempts of selected methods of the derived product. We identify challenges that need to be solved to allow scalable specification and verification of variable software.

1 Introduction

One of the biggest saving potentials for increasing the efficiency of software development lies in the reusability of software artefacts. In order to make software artefacts reusable, two essential qualities must be achieved: flexibility and abstraction. The first is needed, because reusable software is supposed to work in a variety of different contexts and requirements. The second is important to achieve a separation between the level of design and that of executable products. There is a large number of suggestions on how to achieve reusability. Among the most systematic approaches are model-driven engineering (MDE) and software product families (SWPF).[1] Of these, software product families are arguably the more successful method in practice and are very widely used in industry.[2]

The core idea of software product families is to split software development into two separate streams called *Family Engineering* and *Application Engineering*, see Fig 1. In the former, the commonalities of all anticipated products are specified in a structured manner centered around the notion of a *feature*. The resulting interfaces, libraries, and partial implementations are collected in an *Artifact Base*. Concrete products are obtained by feature selection and feature instantiation.

* This work has been supported by the EU project FP7-ICT-2007-3 HATS *Highly Adaptable and Trustworthy Software using Formal Methods.*

[1] Both terms "Software Product Families" and "Software Product Lines" are in use and can be considered to be equivalent within the scope of the present paper.

[2] See the Software Product Line Hall of Fame at http://www.splc.net/fame.html.

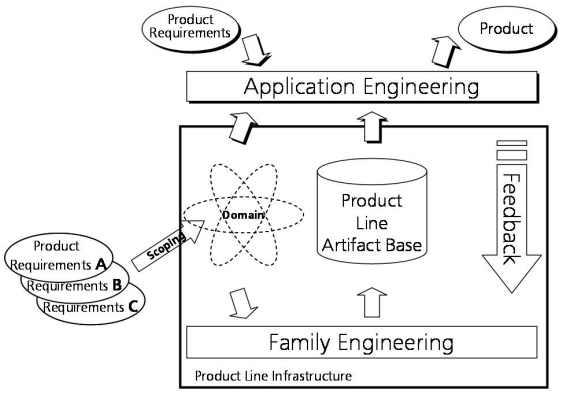


Fig. 1. Sketch of life cycle in Software Product Family development

Software product family-based development is increasingly used in safety-critical applications, for example, in health care products or in automotive software. In addition, the designs of product families reach complexity limitations, because different features may interact in unanticipated ways. It is fairly standard to automatically check the compatibility of features [1], but this is done with structural descriptions, not based on a precise behavioral model of feature functionality. For these reasons it is highly interesting to investigate formal verification of functional properties in software product families. To the best of our knowledge this has not been attempted before. The present paper is a first case study where we seek to verify certain functional properties of a small product family. We report on our experiences, discuss different design choices, and list a number of encountered problems. We also state a number of requirements for the design of verification methods and tools to scale up to industrial-size software with high variability.

It is clear start that—unless verification and specification is compositional and incremental—full functional verification at the family engineering level is doomed to fail, because of the targeted variability. Already small product families give rise to an infeasibly large number of products with different properties. Suitably compositional and incremental verification methods are the subject of future research, therefore, in our case study we aimed at verification at the level of a single derived product in the implementation language Java. At first sight this seems to be merely a standard verification problem. Depending on the implementation of feature selection, however, it becomes much harder: the reason for this is that we chose an implementation that resolves variability points only at run-time, not statically at compile-time. The reason for this choice is that it allows for a more flexible architecture and is, therefore, favored in practice.

The case study in our paper has been done with the verification system KeY [2]. The software product family was implemented in Java and we used the Java Modeling Language (JML) to specify properties.

The paper is organized as follows: in Sect. 2 we describe our case study and provide some background on feature modeling. In Sect. 3 we present the Java implementation of our case study. The formal specification of properties for our case study is explained in Sect. 4, specifically, how we translate FDL into JML. The results of the verification experiments are presented in Sect. 5. We discuss related and future work in Sect. 6.

2 Background

2.1 The Common Component Modeling Example

The Common Component Modeling Example (CoCoMe) [3] is an academic case study and has been widely used as a benchmark for the evaluation of modeling formalisms in the context of software product families (SWPF).

The CoCoMe scenario describes a trading system as it may be used in a supermarket. The basic components of the trading system are cash desks (see Fig. 2) and a store server. A cash desk is responsible to register the products a customer is going to buy. Each product is uniquely identified by a product identification number (productID). During product registration, the cash desk queries the store server for the product name and price tag associated to the entered productID. After all productIDs of the customer's purchase are registered, the customer is accounted for the purchase. Finally, if the payment transaction is successful the store server records the purchase and updates its inventory list accordingly.

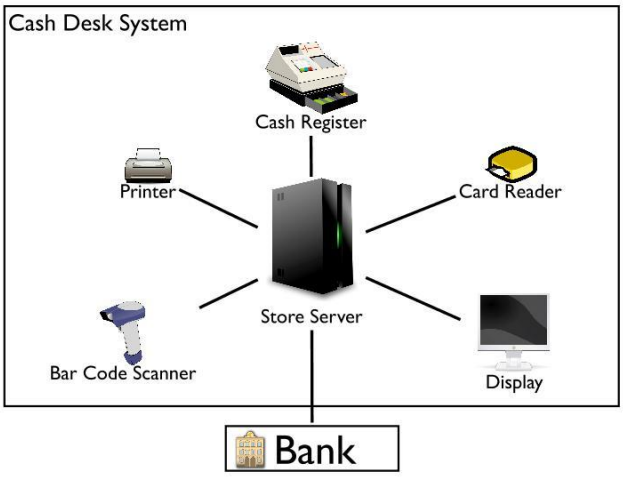


Fig. 2. The hardware components of a single cash desk. Image adapted from [3].

The CoCoMe challenge comprises not only to model a cash desk system that is able to handle the above scenario, but the modeled trade system should also be adaptable to different environments. For instance, shops may have barcodes encoding the product id and want to be able to read them automatically using a scanner rather than having to enter them manually. Businesses may accept only cash or card or both payment kinds. In case of card payments the supported type of cards (prepaid card or credit card) should also be customizable.

2.2 Feature Modeling

In this section we present the specific feature model used for the case study and explain the necessary elements of the feature modeling language. As a basis for our case study we used the feature model in [4] which we also took as a starting point for our implementation.

Different modeling formalisms have been developed to capture the requirements as sketched in Sect. 2.1 in a structured manner. Best known are perhaps decision diagrams [5] and feature-based models [6, 7]. For our case study we use the feature modeling approach first introduced in Feature Oriented Domain Analysis (FODA) [6] and extended in subsequent work.

A software family can be seen as a set of features, while a concrete software product is then derived by selecting a subset of the features; such a feature selection is called a *feature configuration*. Not each combination of features represents a valid feature configuration, as for example, certain features may require the presence or absence of others. Feature diagrams and a feature description language (FDL) provide structured means to describe valid feature configurations. We restrict ourselves to tree like structures for representing valid feature configurations. The feature diagram representing all valid configurations of the feature CashDesk is shown in Fig. 3.

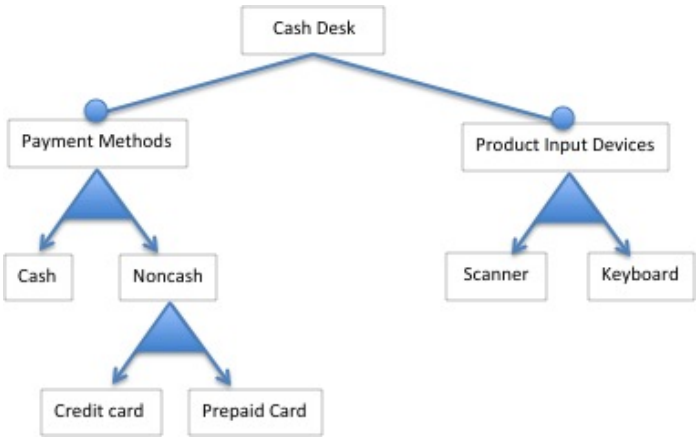


Fig. 3. Feature diagram of the CoCoME cash desk feature

The root of a feature diagram represents the top-level feature whose valid configurations are modeled (here, the cash desk component). A feature can then be composed of subfeatures represented as children of the root node (e.g., the Cash Desk has two subfeatures, namely Payment Method and Product Input Devices). There are different types of edges (see Fig. 4) that can be used to connect the children to its parent. Depending on the type of edge certain restrictions apply. An edge with a filled circle at the end represents a mandatory feature, i.e., the feature *must* be selected when its parent is selected, while an empty circle represents an optional feature. To express that at least one of a group of sibling features has to be selected, the edges to these siblings are connected by a filled triangle. An empty triangle means that *exactly* one of the grouped siblings has to be selected, but not more.

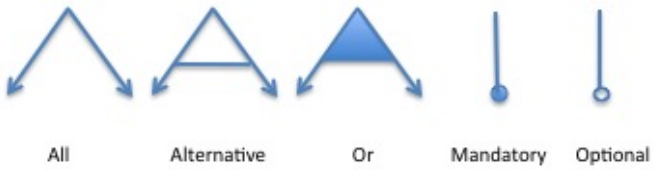


Fig. 4. Feature Diagram Notations: All: all subfeatures must be selected; Alternative: exactly one subfeature must be selected; Or: at least one subfeature must be selected; Mandatory: required feature; Optional: optional feature

In our case a valid configuration of a CashDesk must include the direct subfeatures Payment Methods and Product Input Devices. The feature Payment Methods requires at least one of the features Cash or Noncash to be present.

The most basic product that can be derived from a feature configuration that is valid under the model given in Fig. 3 is the one that allows only keyboards as product input devices and accepts only cash payment.

Alternative to the graphical notation, equivalent textual notations can be used to encode valid feature configurations. We presented only those notions required for the understanding of the paper: there exist several others that allow to express further dependencies and restrictions of features.

3 Implementation

In this section we describe the Java implementation of the cash desk component. We explain how the variability of the cash desk component is achieved so that for all feature configurations described in Fig. 3 a corresponding product can be derived.

The implementation follows closely the feature diagram shown in Fig. 3. For each node there is a similarly named interface or class that represents or implements the feature. The class `CashDesk` shown in Fig. 5 implements the

behavior common to all possible cash desk configurations. A cash desk can be equipped with an arbitrary number of input devices and payment processes. It provides, therefore, methods to add input devices `addInputDevice(IDevices)` and payment methods `addPaymentMethod(IPayments)`.

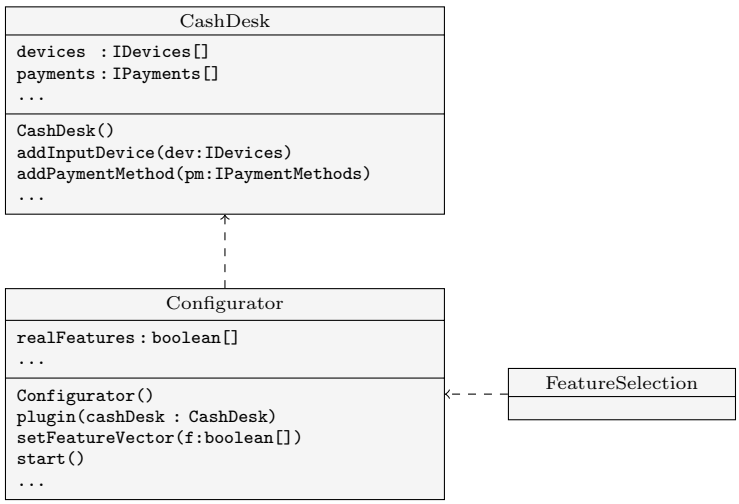


Fig. 5. Class diagram of `CashDesk` controller

Each input device has to implement the interface `IDevices`. The interface `IDevices` defines the protocol for entering product identification numbers by declaring a common set of methods initiating and finalising the product input process. In our scenario, the supported input devices are keyboards (class `KeyboardProductInput`) and barcode scanners (class `ScannerDevice`) as shown in Fig. 6. Supported payment methods need to implement the `IPayments` interface which defines the common protocol for financial transactions. It provides the `CashDesk` class to implement billing of the customer in a transparent way with respect to the underlying low-level payment protocol.

Our implementation of variability points is substantially different to the CoCoMe implementation in [4] and is an almost complete rewrite of it except for the graphical user interface. In principle, our implementation admits to change the feature configuration of an already deployed system at run-time. This means that resolution of variability points happens dynamically rather than statically. In our case study, however, dynamic variability point resolution is not exploited, but restricted to simulate static resolution. Thus, once the system has been setup, its configuration is considered to be fixed. The dynamic evolution of features after system initialisation are beyond the scope of this paper and subject of future work.

We explain now how feature selection and the initialisation of the cash desk system are implemented. At start of the configuration phase the user is asked to

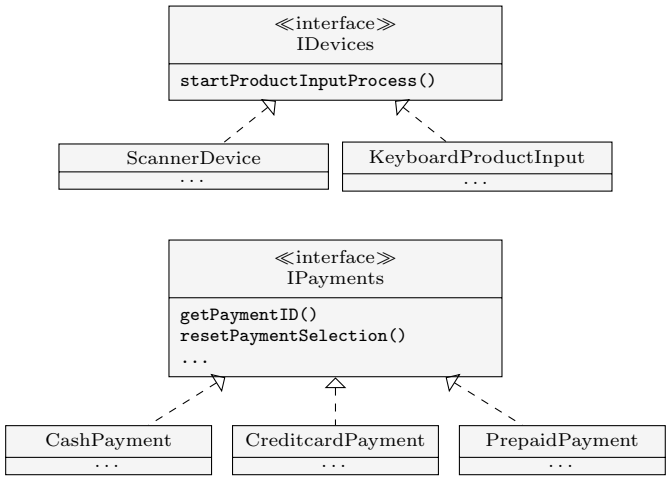


Fig. 6. The feature hierarchy as implemented in Java

customize the system by selecting a feature combination with help of a graphical user interface (see Fig. 7). When the user finished feature selection the chosen

Fig. 7. Feature Selection Interface

configuration is passed to an instance of the `Configurator` class which is responsible for the cash desk system deployment phase (see Fig. 5).

The feature configuration is passed as a bitvector (represented as boolean array) to the method `setFeatureVector(boolean[] f)`. The encoding of feature configurations as bitvectors is canonical: the length of the vector is the same as the number of available features and each bitvector element represents exactly one feature (feature f_i is selected iff `f[i]==true`). If the selected feature configuration is invalid, then the configuration phase is aborted and an exception of type `FeatureException` thrown. Otherwise the feature array is assigned to the field `realFeatures`. Subsequent invocation of the `start()` method triggers creation and initialisation of the cash desk system.

First an instance of the class `CashDesk` is created. Then the `plugIn()` method of the `Configurator` is called which equips the created `CashDesk` instance with

the chosen features and accessories like keyboards or scanners by creating the respective instances and registering them at the `CashDesk` instance. The presence of this plug-in mechanism makes dynamic feature selection principally possible.

4 Specification

Feature model diagrams provide a high-level, structural specification of valid feature configurations, but do not relate to a concrete implementation, that is, the actual behavior of a software product family. As we want to verify that the Java implementation described in Sect. 3 permits only valid configurations to be deployed, we need to connect the feature model specification and the actual Java implementation.

4.1 The Java Modeling Language

For Java programs the Java Modeling Language (JML) [8] is widely used as specification language. JML follows the design-by-contract paradigm and is supported by numerous tools like the Java verification system KeY [2] used here.

JML specifications are added as comments to Java source code. They start either with `//@ ....` or are enclosed in `/*@ ... @*/`. Among other things, JML allows to specify invariants

```
//@ public invariant bExp;
```

method contracts for the normal behavior case

```
/*@ public normal_behavior
  @ requires ⟨bExp_req⟩;
  @ ensures ⟨bExp_ens⟩;
  @ assignable ⟨store_ref_list⟩;
  @*/
```

and for the exceptional behavior case

```
/*@ public exceptional_behavior
  @ requires ⟨bExp_req⟩;
  @ signals (Exception e) ⟨bExp_ens(e)⟩;
  @ assignable ⟨store_ref_list⟩;
  @*/
```

where

- the **requires**/**ensures** keywords followed by a boolean JML expression $\langle bExp_{req|ens} \rangle$ represent the method's pre-/postconditions
- the **assignable** keyword followed by a list of store references (fields, array components) specifies the locations that might be at most changed by the method
- the **signals** keyword specifying the postcondition in case that an exception of the indicated type has been thrown.

JML expressions are a superset of Java expressions with a number of additional operators including

- the boolean operator `==>` denoting logical implication
- universal and existential quantifiers

  `(\forall` T i; $\langle bExp(i)_{guard}\rangle$; $\langle bExp(i)\rangle$`)`;
  `(\exists` T i; $\langle bExp(i)_{guard}\rangle$; $\langle bExp(i)\rangle$`)`;

  where the second semicolon means implication in the universal case and conjunction in the existential case.

Finally, we mention ghost and model fields, declared similar to standard Java fields. A ghost field declaration such as `//@ public ghost int i = 5;` declares an integer typed field named `i` and initialises it with the value `5`. Ghost fields have nearly the same meaning as standard fields and can be assigned values within method body statements using the JML **`set`**-primitive `//@ set i = 10;`.

Model fields can be referred to like standard fields in JML specifications, but it is not possible to assign them a value directly as is the case for ghost fields. Typically, they are used in interfaces where they are related to an (abstract) datatype and used to specify interface methods in terms of model fields and the operations its type provides. Implementing classes of the interface express then how their implementation relates to the model field by providing a **`represents`** clause mapping their internals to the model field. For more details see [8].

4.2 JML Representation of the Feature Model

We describe how a feature model is translated into an equivalent JML specification. The obtained JML specification will be self-contained and independent of a concrete implementation. Our translation of feature models into JML follows the approach presented in [9] for propositional logic.

Let FM denote the feature model to be translated and $\mathcal{F} = \{f_0, \ldots, f_n\}$ the set of all its features. The translation $tr(FM)$ of the feature model consists of:

- A model field declaration

  ```
  //@ model public nullable boolean[] feature;
  ```

  including an invariant stating that the length of the `feature` array is equal to the number of features declared in FM.
- A sequence of ghost field declarations

  ```
  //@ ghost public final static int f_0 = 0;
                          ⋮
  //@ ghost public final static int f_n = n;
  ```

  Each ghost field declaration `f_i` defines a compile-time constant associating the corresponding feature f_i uniquely with an array component of the previously declared model field `feature` such that feature f_i is selected iff `feature[f_i]==true`.

– A set of conjunctively connected boolean JML expressions $\mathsf{Inv} = \{e_0, \ldots, e_n\}$ such that each expression encodes the relationship between a feature and its immediate subfeatures.

The conjunction of the JML expressions in Inv encodes the *FM* diagram (recall that we only consider *FM* models being trees). Wlog. we describe now the construction of the JML expression $e_0 \in \mathsf{Inv}$ encoding the relationship between the parent feature f_0 and its children $f_1, \ldots, f_m$: e_0 is the conjunction of

1. `feature[f_i]==>feature[f_0]` (for all $0 \leq i \leq m$) encoding the ancestor link
2. `feature[f_0]==>feature[f_i]` for each mandatory feature f_i
3. ```
   feature[f_0] ==>
      (feature[f_1] &&!feature[f_2] && ... && !feature[f_k])
       || ... ||
      (!feature[f_1] && ... && !feature[f_(k-1)] && feature[f_k])
   ```
   for each alternative relationship between parent and a subgroup of its children $f_l, \ldots, f_k$ where $1 \leq l < k \leq n$)
4. Analogous expressions for the remaining parent-child relationships.

Based on this definition, we implemented an automatic translation from feature models to a JML specification fragment to be used as part of JML invariants and method specifications.

4.3 Connecting Specification and Implementation

While the specification generated in Sect. 4.2 describes all valid feature configurations, it is not yet connected to the actual implementation of the cash desk system. In this section we explain how to relate the feature vector used in the specification to the implementation. The generated feature specification is used to ensure that

1. the `Configurator` accepts only *valid* feature configurations;
2. the `CashDesk` system built by the `Configurator` has all components required by the selected feature configuration.

We start with item 1. In a first step the model and ghost field declarations from above are inserted into the `Configurator` class. In addition, we need to add the invariant *Inv*, however, since the invariant can only be expected to hold after the feature vector is determined and initialized we add a guard that ensures it:

`//@ public invariant !feature == null ==>` *Inv*

Next, the model field `feature` is related to the actual implementation by adding a JML `represents` clause defining how the model field can be mapped to concrete Java constructs. This mapping is trivial and simply states that `feature` is represented by the field `realFeatures` of class `Configurator`.
   ```

The JML semantics says that each non-helper method preserves all invariants, so our specification expresses already that `setFeatureVector(boolean[])` may only accept valid feature configurations. It is straightforward to construct a normal behavior method specification for `setFeatureVector(boolean[] f)`: simply rename `feature` in e with the method parameter `f` and use `feature == f` as postcondition, the only a valid configuration is actually accepted.

Moving to item 2. above, the `feature` array needs to be more closely related to the underlying Java implementation. This can be done in a systematic manner by annotating the Java feature model FM with a mapping that maps each feature f_i to a JML expression $\phi(f_i)$ to be used as an additional invariant to be established after deployment of the product and preserved thereafter. For example, an annotation ensuring that the created cash desk `cashDesk` is equipped with a properly registered keyboard is:

```
feature != null && feature[_keyboard] ==>
 (\exists KeyboardProductInput kpi;
    (\exists int i; 0<=i && i < cashDesk.stateChangeListenerSize;
             cashDesk.stateChangeListener[i]==kpi) &&
    (\exists int j; 0 <= j && j < cashDesk.devicesSize;
             cashDesk.devices[j] == kpi))
```

5 Verification

We used the KeY verification system [2] to prove that the feature configuration validity check and the cash desk system setup procedure are implemented faithfully with respect to the specification given in Sect. 4.

We were in particular interested how well a current state-of-the-art verification tool scales when verifying highly adaptable software as developed in the context of software product families.

```
public void plugIn(CashDesk cashDesk) {
   if (realFeatures[SCANNER]) {
     final ScannerDevice scanner = new ScannerDevice(cashDesk);
     cashDesk.addInputDevice(scanner);
     cashDesk.addStateChangeListener(scanner);
   }

   if (realFeatures[NONCASH] && realFeatures[CREDITCARDREADER]) {
               ...
   }
   ...
}
```

Fig. 8. If-cascade implementing the cash desk initialisation logic

Before we could start the verification of our CoCoMe subsystem, we had to adapt the derived JML specification slightly. The reason is that KeY's support for model fields is somewhat rudimentary. Thus we decided to replace the `feature` model field by a ghost field of the same name. As the semantics of model fields is much more complex than that of ghost fields, we had also to change and extend JML specifications referring to the model field to achieve an equivalent and correct specification. Such a replacement is not possible in general but worked here well in our context due to the simple `represents` clause and by assuming a closed system, i.e., that all classes implementing input devices and payment methods are known in advance.

We were able to verify the correctness of the validity check and most parts of the actual cash desk creation and initialisation. In its original version the latter had been a monolithic method (`plugIn()` of class `Configurator`) consisting of if-cascades as shown in Fig. 8. Verification of this method was infeasible as the proof size exploded. We modularised the monolithic method and separated each if-cascade representing the creation and registration of a device or payment method into different methods. The specification of one of these methods `checkScanner(CashDesk)` is given in Fig. 9.

```
/*@
  @ public normal_behavior
  @ requires feature!=null;
  @ requires feature[_scanner];
  @ ensures
  @  (\exists ScannerDevice sd; \fresh(sd);
  @    (\exists int i; 0<=i && i< cashDesk.stateChangeListenerSize;
  @       cashDesk.stateChangeListener[i]==sd) &&
  @        (\exists int j; 0<=j && j< cashDesk.devicesSize;
  @                                      cashDesk.devices[j]==sd));
  @ assignable
  @  \object_creation(ScannerDevice),\object_creation(Scanner),
  @  cashDesk.stateChangeListenerSize, cashDesk.stateChangeListener,
  @  cashDesk.stateChangeListener[cashDesk.stateChangeListenerSize],
  @  cashDesk.devicesSize, cashDesk.devices,
  @  cashDesk.devices[cashDesk.devicesSize];
  @*/
```

Fig. 9. Specification of the `checkScanner` method

Afterwards we were able to verify most of the individual methods in isolation. Fig. 10 shows statistics about the performed proofs (all fully automatic) and their size. For two methods, `checkCreditCard` and `checkPrepaidCard` we could not yet obtain proofs. We are currently analyzing the problems and we are confident that we can present proofs in the final version of this paper.

Method	Nodes	Branches
checkScanner	22032	107
checkCash	14150	77
checkKeyboard	15755	64

(a) Ensure Postcondition

Method	Nodes	Branches
checkScanners	40439	429
checkCash	20265	161
checkKeyboard	46392	664

(b) Correct Assignable Clause

Method	Nodes	Branches
checkScanners	89492	703
checkCash	49421	327
checkKeyboard	70962	485

(c) Preserve Invariant

Fig. 10. Proof Statistics

6 Related & Future Work

Related Work. In [10] the authors describe an approach to open system verification of software product lines by parametrised interfaces. The verification technology is (3-valued) model checking. Features and the core product are equipped with interfaces externalising certain states as input and output states. Features can be composed to complex features or to whole products by connecting to the core product using these interfaces.

The authors aim to allow compositional (contract-based) reasoning using model checking by computing subcontracts for the interfaces. When composing the features to a complete product only the subcontracts have to be discharged. Specifically, for a given global property of a product, constraints to be posed onto the exposed input and output states are computed independently for each feature. At composition time one has then only to ensure that these constraints are satisfied by the preceding/succeeding features. The constraints for the preceding features are propositional formulas restricting the values of the input values, while those of the succeeding features are temporal logic formulas ensuring that certain properties are adhered to in the future. The presented approach allows to compose products arbitrarily and eases verification by having only to discharge the computed constraints for the derived product, but is limited to incremental features.

The authors of [11] describe an approach to verification of a software product lines based on ASMs and the AHEAD methodology. Their case study is built upon the Jbook [12], where a complete virtual machine for Java 1.0 has been modelled including an interpreter and compiler. The compiler was proven correct wrt. the interpreter.

The authors restructured the Jbook case study to fit into the feature modelling approach as enforced by the *feature-oriented programming* (FOP) design

methodology which provides a technology for compositional program assembly. The so obtained structure has a base layer or core representing only a subset of Java expressions (imperative expressions). This core is stepwise extended by adding new features (layers) such as imperative statements, class and object features until complete coverage of the Java 1.0 language is reached.

In this framework a strong structural connection exists between model extension and the correctness proof of the compiler. This allows to alter the existing correctness proof by adding new independent cases (e.g., for new supported language constructs) or to refine existing cases by an additional invariant to be proven. Features having a non-compositional or destructive influence for existing cases occurred either rarely or not at all. Correctness has been proven (mostly) by hand without any automation or even machine-checked proof support.

Future Work. We differ substantially in our objectives and the underlying technology from the work discussed above: we aim at a highly automatised compositional design and verification system that is applicable to adaptive systems in general and specifically to software product line engineering. In basing our work on an expressive program logic and specification framework realized in a verification system with a high degree of automation we overcome some principal limitations, however, we are fully aware that there are considerable research challenges ahead:

- We believe that it is not sufficient to achieve compositionality by manually adding case distinctions or refining existing ones. The verification system and methodology must inherently construct proofs that are accessible to compositional reasoning and—where this is not possible—apply proof reuse techniques.
- Support for destructive features is essential: the restriction to mostly incremental features and consequently conservative extensions is not sufficient for our purposes. Adding new features may easily render existing proof cases invalid and require a completely new proof. Again, proof reuse is of essence.
- Independence of new features and existing proofs: even if a new feature has no influence on, say, a certain class invariant, this needs to be proven (or enforced) explicitly. In case of real-world languages like Java with aliasing this is still an area of active research [13, 14].

Our case study showed that formal specification and verification of software product families is, in principle, possible with current technology and can actually be achieved for small examples. Nevertheless, the results of our case study are not satisfactory from our point of view. Specific problems, such as missing support for model fields which are crucial for verification of open systems, are specific shortcomings of the used verification tool KeY and will be resolved in the near future. Others issues, however, such as proof-size explosion due to the resolution of variability points needs to be solved on a methodological level. Research regarding this issue is under way. It would also be interesting to explore

how separation-logic based approaches perform in the context of software families and if they can overcome some of the problems we faced because of framing issues.

References

1. Benavides, D., Segura, S., Ruiz-Cortés, A.: Automated analysis of feature models 20 years later: a literature review. Information Systems (2010)
2. Beckert, B., Hähnle, R., Schmitt, P., eds.: Verification of Object-Oriented Software: The KeY Approach. Volume 4334 of LNCS. Springer-Verlag (2007)
3. Rausch, A., Reussner, R., Mirandola, R., Plasil, F., eds.: The Common Component Modeling Example: Comparing Software Component Models [result from the Dagstuhl research seminar for CoCoME, August 1-3, 2007]. In Rausch, A., Reussner, R., Mirandola, R., Plasil, F., eds.: CoCoME. Volume 5153 of LNCS., Springer (2008) Preliminary version of the chapter describing the Trading System is available at: http://agrausch.informatik.uni-kl.de/CoCoME/downloads/documentation/cocome.pdf.
4. Schaefer, I., Worret, A., Poetzsch-Heffter, A.: A Model-Based Framework for Automated Product Derivation. In: Proc. of Workshop in Model-based Approaches for Product Line Engineering (MAPLE 2009). (2009)
5. Bayer, J., Gacek, C., Muthig, D., Widen, T.: Pulse-i: Deriving instances from a product line infrastructure. Engineering of Computer-Based Systems, IEEE International Conference on the **0** (2000) 237
6. Kang, K.C., Cohen, S., Hess, J., Nowak, W., Peterson, S.: Feature-Oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-021, Carnegie Mellon University Software Engineering Institute (1990)
7. Batory, D.: Feature models, grammars, and propositional formulas. In: Software Product Lines. Springer-Verlag (2005) 7–20
8. Leavens, G.T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., Müller, P., Kiniry, J., Chalin, P., Zimmerman, D.M.: JML Reference Manual. (September 2009) Draft revision 1.235.
9. Czarnecki, K., Wasowski, A.: Feature diagrams and logics: There and back again. In: Software Product Line Conference, 2007. SPLC 2007. 11th International. (2007) 23–34
10. Blundell, C., Fisler, K., Krishnamurthi, S., Hentenryck, P.V.: Parameterized interfaces for open system verification of product lines. In: ASE '04: Proceedings of the 19th IEEE international conference on Automated software engineering, IEEE Computer Society (2004) 258–267
11. Batory, D.S., Börger, E.: Modularizing theorems for software product lines: The jbook case study. J. UCS **14**(12) (2008) 2059–2082
12. Stark, R.F., Börger, E., Schmid, J.: Java and the Java Virtual Machine: Definition, Verification, Validation with Cdrom. Springer (2001)
13. Müller, P.: Modular Specification and Verification of Object-Oriented Programs. Volume 2262 of Lecture Notes in Computer Science. Springer (2002)
14. Schäfer, J., Reitz, M., Gaillourdet, J.M., Poetzsch-Heffter, A.: Linking programs to architectures: An object-oriented hierarchical software model based on boxes. In: CoCoME. (2007) 238–266

Vótáil: PR-STV Ballot Counting Software for Irish Elections

Dermot Cochran and Joseph R. Kiniry

IT University of Copenhagen, Denmark
(dero,kiniry)@itu.dk

Abstract. *Vótáil* is an open source Java implementation of Irish Proportional Representation by Single Transferable Vote (PR-STV). Its functional requirements, derived from Irish electoral law, are formally specified using the Business Object Notation (BON) and refined to a Java Modeling Language (JML) specification. Formal methods are used to verify and validate the correctness of the software. This is the first public release of a formally verified PR-STV open source system for ballot counting and the most recent of only about half a dozen releases of formally verified e-voting software.

1 Introduction

Vótáil is the Irish Gaelic word for Vote. Many aspects of the election process are apparently suitable for automation. For example, voter registration records are stored in computer databases, postal voting has sometimes been replaced with internet voting, paper ballots with voting kiosks, and ballot counting is sometimes done by machine. However, many attempts to introduce electronic voting have failed, or at least received much criticism, due to a litany of software and hardware errors, many of which are avoided through the use of formal methods.

To give evidence to this claim, as well as to continue to play a scientist-activist role in the public eye, we present the *Vótáil* system: a verified implementation of Irish PR-STV. This work in novel on several fronts. First, *Vótáil* represents one of the largest and most complex case studies in the verification of an object-oriented system. As such, it helps validate our verification-centric approach to software design and implementation [15,16]. Second, as the case study is directly relevant to one of *society's* critical systems, it represents an opportunity to influence the mindset of not just the software developer interested in quality, but in the worldview of the politician interested in trusted elections and the voter passionate about their government. Finally, this work also pushes some of the object-oriented software verification's community's tools to their limits, thus shows us where we have succeeded, and where we have failed, in our focus on usable, usable, and powerful verification tools. Consequently, we expect that this system, like others of a similar ilk (e.g., the KOA tally system [11,14]) will be used to benchmark new verification tools for object-oriented software.

Before discussing the process, tools, and techniques used in this case study, some context in election system and voting in Ireland is necessary to appreciate its size and complexity.

1.1 Electronic Voting in Ireland

In 2009, the Irish government decided to save costs by disposing of its current generation of direct recording electronic (DRE) voting machines. The decision to stop using electronic voting was due to technical problems and to more general concerns about the security of electronic voting. The current political consensus is that electronic voting (in its current form, e.g., DRE machines) will not be used in the Republic of Ireland.

In the authors' personal experience, Irish citizens are happy with the paper-based voting process and the hand-counting of votes that takes several days with gradual reporting of results through the media. There are usually one or two closely contested seats that require a full manual recount with additional observers.

After opposition parties voiced concerns about electronic voting, the Irish Government established an independent ad-hoc Commission on Electronic Voting (CEV). One of the recommendations of the CEV was the use of a Voter Verified Paper Audit Trail (VV-PAT) [10]. A VV-PAT is a printed copy of the electronic ballot which is used for manual recounts and audits of the result. However, the cost of adding a VV-PAT as well as other improvements to the software of the voting machines was seen as prohibitive, making it more economical for Ireland to abandon the use of electronic voting.

1.2 Voting Scheme

The Republic of Ireland uses Proportional Representation by Single Transferable Vote (PR-STV) for its national, local and European elections. PR-STV is a ranked choice voting system, in which each voter ranks the candidates from first to lowest preference. A quota is the minimum number of seats needed to win one seat. If a candidate has more than the quota, the surplus votes are transferred pro-rota to the next highest preference on the ballot. If not enough candidates have a quota, then the lowest candidate is excluded and his or her ballots transfered to the next highest preference.

Oireachtas Éireann, the National Parliament of the Republic of Ireland, has two chambers. The people directly elect Dáil Éireann, the lower chamber of the Oireachtas, for a term of up to five years by a quota-based single transferable vote system in multi-seat constituencies. The upper house, called the Seanad, also uses PR-STV, but uses postal ballots and is indirectly elected, except for the six seats elected by university graduates. The Seanad has an advisory role and a smaller electorate. It is therefore a lesser target for electoral fraud, and a low risk election, so it is more interesting and important to look at verification of Dáil elections.

Note that Irish legislation uses the term 'vote' as a noun to mean the contents of a ballot paper rather than as a verb for the action of casting a ballot [9]. For the purpose of clarity, *vote* means the full set of candidate preferences recorded by a voter at an election.

The political significance of lost, corrupted or altered votes depends on the type of voting system (e.g., STV) and the closeness of the election race. In PR-STV, it is not unusual to see the final seat in a multi-winner constituency determined in the last round of counting by a small number of votes.

Manual recounts are often called for closely contested seats, as the results often vary slightly, indicating small errors in the manual process of counting votes. Paper-based voting with counting by hand is popular in Ireland, and recent attempts at automation were frustrated by subtle logic errors in the ballot counting software [6]. The logic errors exist, in part, due to the complexities and idiosyncrasies with regard to tie breaking, especially involving the rounding of vote transfers. Other errors relate to the rounding up or down of ballot transfers and to the randomisation effect of ballot shuffling, which does not have have a precise legal definition. As every ballot and every preference on a ballot can make a difference when the last seat of a multi-seat constituency is being decided, these subtle errors can have an enormous effect on the outcome of an election.

There has been some desire in Ireland to simplify matters. Referenda to introduce plurality (first past the post) voting were rejected twice by the Irish electorate, in 1959 and again in 1968 [19]. Since then, there have been no further legislative proposals to change the voting scheme used in Ireland.

The following are selected quotes from the CEV report on the previous electronic voting system used in Ireland [10]:

> – Design weaknesses, including an error in the implementation of the count rules that could compromise the accuracy of an election, have been identified and these have reduced the Commission's confidence in this software.
> – The achievement of the full potential of the chosen system in terms of secrecy and accuracy depends upon a number of software and hardware modifications, both major and minor, and more significantly, is dependent on the reliability of its software being adequately proven.
> – Taking account of the ease and relative cost of making some of these modifications, the potential advantages of the chosen system, once modified in accordance with the Commission's recommendations, can make it a viable alternative to the existing paper system in terms of secrecy and accuracy.

Thus, Ireland wishes to keep its current complicated voting scheme, is critical of the existing attempts to implement that scheme in e-voting, but keeps the door slightly ajar for the introduction of e-voting in the future. Consequently, we believe that this combination of factors makes our work timely, relevant, and, potentially, high-impact. In the end, our meta-goal is to show that, if a handful of researchers working in their spare time can design and implement a

verified voting system for one of the most complex voting schemes in the world, citizens and governments must *demand* that their e-voting systems are of at least this level of quality. *Verified elections effected, in part, through formally verified voting software are mandatory for future e-democracies.*

1.3 Related Work

The authors are unaware of any peer-reviewed published related work on the formal specification and implementation of PR-STV. We are aware of some unpublished or unfinished work relating to previous attempts at formalization of PR-STV, including some Prolog work by Naish and an implementation of the Scottish STV system in CLEAN by researchers at the Radboud University Nijmegen. The only peer-reviewed published related work of interest is a protocol for the tallying of encrypted STV ballots [20] and verifying properties of voting protocols, not software (e.g., several papers by Ryan [7]).

There have been numerous pieces of work on contract-guided or refinement-centric software verification, particularly from the correctness-by-construction community. Only a few focus on the particular challenges inherent in modern object-oriented systems (e.g., the work of Nunes and colleagues [18]), and none that we are aware of include support for traceable refinement from requirements and features to verified software.

Finally, some work on the use of logics to understand and reason about law is relevant, e.g., the work of van der Meyden has influenced us [21]. We do not attempt to use such (deontic) logics in our refinement from law to formal models, though doing such may improve the quality and correctness of our specification and its refinement.

1.4 Outline of Paper

The rest of the paper is organized as follows. Section 2 describes our methodology for refinement of requirements from electoral law to Java software. Section 3 contains a summary of the requirements and features demanded for PR-STV ballot counting. Section 4 reviews the formal specification of PR-STV. Next, in section 5, the verification and validation of the software system are detailed. Finally, section 6 concludes the paper with some reflections.

2 Methodology

To appreciate the rigor involved in formally specifying and verifying a ballot counting system for a non-trivial electoral system like PR-STV, discussing details about our methodology is warranted.

2.1 Business Object Notation

Business Object Notation (BON) provides a high-level object orientated description of a system [22]. BON can be thought of as a rigorous subset of UML.

BON has two flavors: informal BON and formal BON. Informal BON looks like a structured natural language, but is checked for well-formedness in a variety of ways. Formal BON looks like a strongly typed object-oriented, parametric class-based programming language with contracts and behavioral specifications. Specifications written in formal BON are essentially semantic dependent types. Refinement from informal to formal BON is described in the aforementioned text and supported by our BONc tool suite[1].

2.2 Java Modeling Language

The Java Modeling Language (JML) is a formal behavioral interface specification language used to specify the behavior of Java software [17]. It extends Java with annotations for specifying simple formal statements in a design-by-contract (DBC) style [2] and model-based specifications a la Larch [1]. Informal BON is either refined to a formal specification in formal BON or directly to a formal object-oriented specification language such as JML. Support for performing and checking such refinements is provided by our Beetlz tool[2].

2.3 A Verification-centric Development Process

A set of functional requirements and features, derived from electoral law, is a semi-formal specification, although written in a structured way. To translate the ballot counting process, as defined by law, into an executable software system we define an abstract state machine (ASM). This ASM and a set of functional requirements (described later) are refined into an object-oriented system design using BON, which is in turn refined into a JML contract-based specification. The JML specification and ASM are then implemented in Java. Thus, we follow a strict design-by-contract based approach to software engineering.

Validation is accomplished via testing. Automated tests are generated from the JML specification, and scenario tests are derived from the ASM. Finally, the entire system is verified using extended static checking, a kind of automated functional verification.

Figure 1 provides an overview of these artifacts and their interrelationships. Details of this process and how these refinements are represented and reasoned about is not the focus on this paper. The interested reader is encouraged to examine our other published work on this front [15,16].

3 Requirements for PR-STV Ballot Counting

In addition to the general requirements for e-voting, like ensuring privacy of the voter and accuracy of the count, electoral-specific requirements are also derived from electoral law and government regulations about the counting of votes. In

[1] http://www.kindsoftware.com/products/opensource/BONc/
[2] http://www.kindsoftware.com/products/opensource/Beetlz/

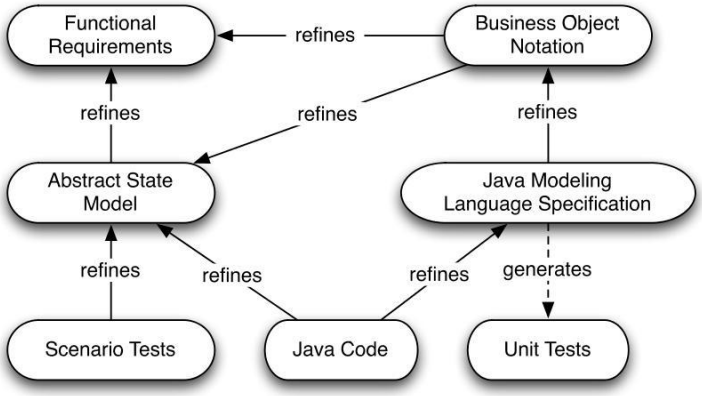


Fig. 1. Relationships between software engineering artifacts.

the Irish context, these requirements come from a commission on voting and electoral law and the electoral act itself.

We insist that a tracable interpretation of these requirements and features from law or similar to a concrete software system is mandatory.

3.1 Quality Requirements

The Commission on Electronic Voting (CEV) laid down several guidelines for development of electronic voting systems, including the following [10]:

- clear definition of functional requirements and specifications
- robust and formal approach to design and development
- separation of critical concerns, e.g., voter registration and ballot counting
- publication or public inspection of the source code (preferably open source)
- open public testing of the system

The pilot e-voting project in Ireland failed to meet any of these criteria, leading to its rejection, whereas Vótáilfulfills all five of the quality requirements listed above.

3.2 Functional Requirements in Electoral Law

The 1992 Electoral Act, including subsequent amendments, and the Commentary on Count Rules issued by the CEV [8], is the starting point for our requirements analysis. In previous work 39 semi-formal statements are used to describe these functional requirements for ballot counting in elections to the Dáil [4].

A few example formal statements from our previous work are listed and cross-referenced, as shown in Table 1. The *section*, *item* and *page* column titles refer to the CEV Commentary on Count Rules, which in turn refers back to the Electoral Acts.

Table 1. Cross-referencing functional requirements and the law.

ID	Functional Requirement	Section	Item	Page
8	If the number of continuing candidates is equal to the number of seats remaining unfilled, or the number of continuing candidates exceeds by one the number of unfilled seats or there is one unfilled seat, then do not distribute any surplus unless it could allow one or more candidates with at least one vote to save their deposits.	4	2	15
9	Not more than one surplus is distributed in any one count.	4	3	16
10	Where there are seats remaining to be filled, but no surpluses available for distribution, the lowest continuing candidate or candidates must be excluded.	4	4	16
11	There must be at least one continuing candidate for each remaining seat.	4	4	16
	...			

4 Formal Specification

The formal specification has several aspects. First, we must formalize the ballot counting process — the steps through which one must pass to convert a pile of legal ballots into a tally. Secondly, we must capture the various stages through which each key element of the counting process (e.g., a candidate, a ballot, a ballot box, etc.) can pass. The formalization of these two different, but interrelated, facets of the specification of are done via the use of ASMs.

4.1 Abstract State Machine

A two tier Abstract State Machine (ASM) is used to represent the 39 functional requirements. The upper tier of the ASM describes the state of the election (`EMPTY`, `SETTING_UP`, `PRELOAD`, `LOADING`, `PRECOUNT`, `COUNTING`, `FINISHED`, `AUDIT`, `REPORT`) in a linear way, in which there is only one possible transition into and out of each state, whereas the lower tier of the ASM (shown in Figure 2) is more complex and describes more detailed sub-states and transitions within the `COUNTING` state.

4.2 Invariants

An invariant is a predicate about a set of objects in the system that must always hold during stable/quiescent states during system execution. In essence, the invariants of an object and its class hierarchy explain what constitutes a valid instance of the object in question. Likewise, invariants about the states of an ASM explain what must be true of the process and the objects on which it operates for the process itself to be valid.

Each election state has a number of invariants that must hold. For example, in the fragment of JML seen in Figure 3, the *Finished* state has as an invariant

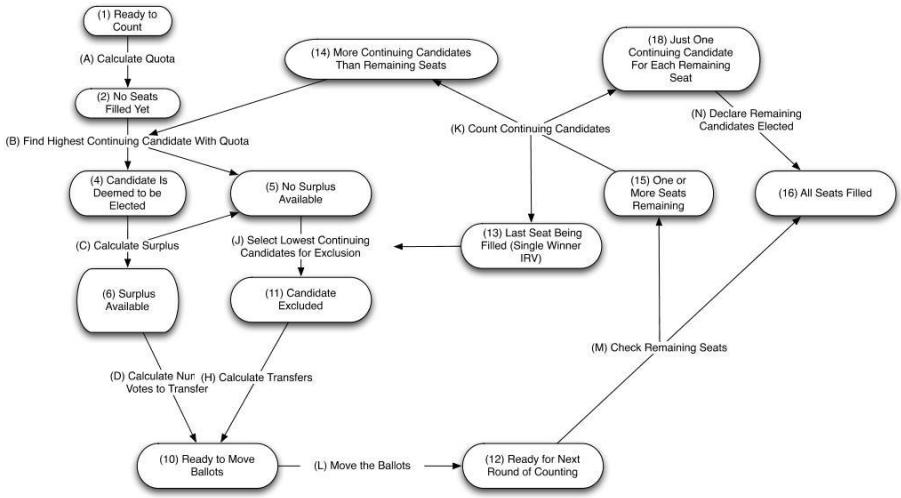


Fig. 2. Abstract State Model, lower tier (sub-states)

that the number of candidates elected equals the number of seats available, whereas the *Pre-Count* state has an invariant that the number of candidates elected (so far) is zero. Invariants are coupled to states in the ASM through a variable `state` denoting the obvious. Some invariants must only hold in a given state and others must hold for that state and all future states, as in the example.

4.3 Coupling State Transitions

Reasoning about the overall correctness of the counting algorithms implementation boils down to reasoning about the top-level ASM. If we can show that each transition in the ASM is valid (it only goes from legal pre-states to legal-post states, as defined by law), then we can guarantee, by transitivity, that the overall algorithm is correct. The correctness of these transitions is captured entirely by invariants, thus an example of how the invariants are preserved in the top-level state transitions is illuminating.

Example 1. Let E be the number of candidates elected (a variable), and let S be the number of seats being filled (a constant), and let R be the number of seats remaining to be filled, a function defined to mean $S - E$. Consider the transition in the example from the *Pre-Count* state to the *Counting* state.

At *Pre-Count*: $E = 0$, and for all candidates C, the status of C is *Continuing*. After transition to *Counting* $E = 0$. The new invariant that must hold is that E equals the number of *Candidate* objects C where C is *Elected*. All candidates have *Continuing* status to begin with, so $E = 0$ and the invariant is satisfied.

Likewise, consider the transition from the *Counting* state to the *Finished* state. Before the transition from *Counting* to *Finished*, the inner state machine

```
/** Number of candidates elected so far */
//@ public model int numberElected;
//@ public invariant 0 <= numberElected;
//@ public invariant numberElected <= seats;
/*@ public invariant (state <= PRECOUNT) ==>
   @   numberElected == 0;
   @ protected invariant (COUNTING <= state) ==>
   @   numberElected == (\num_of int i;
   @   0 <= i && i < totalCandidates;
   @   isElected(candidateList[i]));
   @ public invariant (state == FINISHED) ==>
   @   numberElected == seats;
   @*/
```

Fig. 3. A JML specification describing the number of candidates elected.

must be in the *End-of-Count* sub-state and $S = E$, therefore $R = 0$. The inner state machine can only reach the *End-of-Count* sub-state when $R = 0$. Therefore $E = S$ and the invariant for the *Finished* state holds.

Similar reasoning is used to analyze the correctness of each invariant on each state of the ASM, invariants that span ASM states, as well as the legitimacy of transitions between states.

4.4 Other Examples of Invariants

All invariants must hold in every state, not just those state pairs at the end of transitions in the top-level ASM. These legal invariants are expressed by class and object invariants in the JML specification. Consequently, when a transition between states occurs, the invariants of both the old and new state must hold during the transition (i.e., during any helper methods that are called while the software is moving between states).

4.5 Refinement to BON

To formally capture legal requirements, as expressed through invariants, and to rigorously refine our ASMs into a software system, the architecture of our ballot counting system (i.e., its classifiers and their relations) and its correctness properties (i.e., its invariants) are formally specified in the Business Object Notation.

Each state transition in the Abstract State Model is represented either by a command or a query in BON. In BON, a command is an action that changes the state of an object, for example, moving a ballot from one pile to another, whereas a query returns some information about the system. A query is implemented in JML either as a field with invariants or as a pure method.

The example in figure 4 shows an informal BON description of the Ballot Counting process.

Transitions into	**Invariant for new state**
End of Count	The number of candidates elected equals the number of open seats.
No Surplus Available	All continuing candidates have less than a quota of votes.
No Seats Filled Yet	The number of elected candidates is zero.
Candidates Have Quota	There exists a continuing candidate with at least one quota of votes.
Candidate Elected	The number of elected candidates is less than the number of open seats.
This Candidate Status is Elected	This candidate had at least one quota of votes.
Candidate Excluded	This candidate had fewer votes than any other continuing candidate.
Last Seat Being Filled	The number of elected candidates is one fewer than the number of open seats.
More Continuing Candidates than Remaining Seats	The number of open seats is less than the sum of the number of elected candidates and the number of continuing candidates.
Seats Remaining	The number of elected candidates is less than the number of open seats.
One Continuing Candidate per each Remaining Seat	The sum of the number of elected candidates and the number of continuing candidates is equal to the number of open seats.

Table 2. Examples of invariants for each sub-state, translated from JML to English for the reader.

```
class_chart BALLOT_COUNTING
explanation
  "Count algorithm for tallying of the votes in Dail elections"
query
  "How many continuing candidates?",
  "How many remaining seats?",
  "What is the quota?",
  "Who is/are highest continuing candidate(s) with a surplus?",
  "What is the surplus?",
  "What is the transfer factor?"
command
  "Distribute the surplus ballots",
  "Select lowest continuing candidates for exclusion",
  "Declare remaining candidates elected",
  "Close the count"
end
```

Fig. 4. An Informal BON description of the Ballot Counting class.

4.6 Refinement to JML Specification

The BON design contains 1 cluster[3] with 5 classifiers, 20 queries, 5 command and 6 constraints. These are refined to 1 package with 10 classes, 104 methods, 70 invariants, 192 preconditions and 117 postconditions in JML.

We used a version of JML that extends Java 1.4, because of existing mature tool support. We also minimize our use of the JDK and use simple data structures such as arrays. When using arrays in JML we make assumptions about the maximum number of candidates and the maximum number of ballots, based on past elections and the theoretical maximum population of a constituency.

```
/** Number of votes needed to win a seat */
//@ requires 0 <= seats;
//@ ensures \result == 1 + (totalVotes / (seats + 1));
public /*@ pure @*/ int getQuota();
```

Fig. 5. An example of a JML specification for a BON query.

Two examples of JML are shown in figures 5 and 6, one for a query and one for a command, and are examples of the initial JML specification written during refinement. Such a specification contains only the signature of each method without implementation code (the implementation is "bottom," aka "assert false.").

[3] A BON cluster is a collection of related concepts, roughly similar to a Java package.

```
/**
 * Transfer votes from one candidate to another.
 * @param fromCandidate Elected or excluded candidate
 * @param toCandidate   Continuing candidate
 * @param numberOfVotes Number of votes to be transferred
 */
/*@ requires fromCandidate.getStatus() != CandidateStatus.CONTINUING;
  @ requires toCandidate.getStatus() == CandidateStatus.CONTINUING;
  @ ensures countBallotsFor(fromCandidate.getCandidateID()) ==
  @         \old (countBallotsFor(fromCandidate.getCandidateID()))
  @         - numberOfVotes;
  @ ensures countBallotsFor(toCandidate.getCandidateID()) ==
  @         \old (countBallotsFor(toCandidate.getCandidateID()))
  @         + numberOfVotes;
  @*/
public abstract void transferVotes(
  final /*@ non_null @*/ Candidate fromCandidate,
  final /*@ non_null @*/ Candidate toCandidate,
  final int numberOfVotes);
```

Fig. 6. An example of a JML specification for a BON command.

Note also that the method signature specification in this example states in a precondition that none of the parameters can have null values.

4.7 Architecture

There are 10 Java classes in the implementation, representing the actors in the system, for example Ballots, Ballot Boxes and Candidates. Figure 7 shows the relationship between the Java classes, the `BallotCounting` class contains the specifics of PR-STV, whereas the `AbstractBallotCounting` class contains the more general properties of ballot counting algorithms. Class names shown in italics are supporting classes that were added in the Java implementation but were not refined from BON.

4.8 The Vótáil Theorem

Due to the manner in which the formal specification of the ballot counting algorithms is accomplished (section 3), and the aforementioned argument about the correctness of state transitions (section 4), we summarize via informal refinement the overall theorem expressed by this ballot counting system as the *Vótáil theorem*.

> **Vótáil Theorem:** Given a valid set of candidates up for election for a set of seats, and a ballot box containing a valid set of ballots, after the ballot counting algorithm executes, we guarantee that the candidates deemed elected by Vótáil are exactly those elected by Irish law.

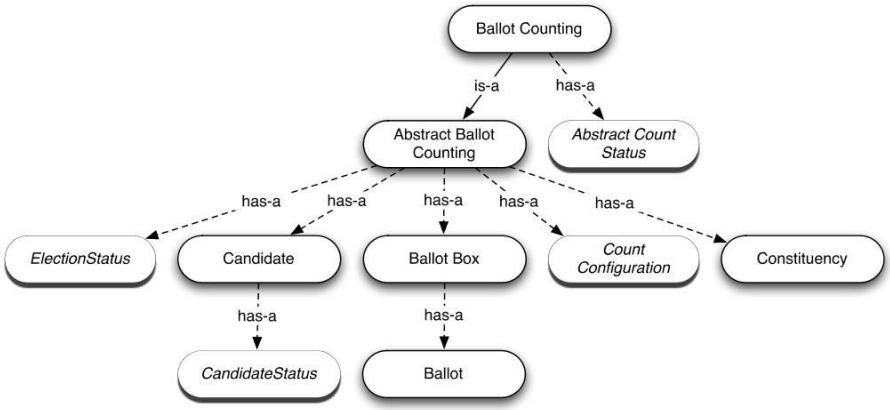


Fig. 7. Relationship between classes

In other words, the candidates elected by a machine count are the same as would be elected in a *correct* manual count of paper based ballots.

To derive the above "theorem," we essentially translate the formal preconditions of the first state of our ASM and the formal postconditions of the last state of our ASM back into English that is digestible by the voter.

5 Verification and Validation

Unlike the vast majority of e-voting systems available today, we have made our system's specification, implementation, validation, and verification available for public review.

5.1 Open Source Implementation

The source code is open source, under the terms of the MIT open source license, and is available via our Trac server[4]. The source code is managed using a subversion server hosted on our website[5] and developed using the Mobius Program Verification Environment (PVE)[6]. The source code contains 723 Java statements in 11 classes and 92 methods.

[4] https://trac.ucd.ie/repos/Votail/tags/0.0.1b
[5] https://trac.ucd.ie/repos
[6] mobius.ucd.ie

5.2 Scenario Tests

Ten hand-written scenario tests are derived from the ASM and provide 97 percent code coverage. To measure coverage we use the EclEmma[7] code coverage plug-in for the Eclipse Integrated Development Environment[8] and run tests using JUnit.

The remaining 3 percent of code is accounted for by non-executable statements, such as declarations of constants that were either not exercised directly by the unit tests, or not measured by the coverage tool. We also run all scenario tests with JML runtime assertion checking (RAC) enabled to double-check their consistency and correctness.

5.3 Automatic Generation of Unit Tests

Just over 7,000 unit tests were generated from the JML specifications. The JMLUnit tool allows the automatic generation of unit tests [3]. JMLUnit requires guidance on which data types and values are interesting for testing and will then generate tests for each precondition, postcondition and invariant for all permutations of the test data. The interesting values of this case study were derived manually via a careful examination of the Vótáil architecture and its legal requirements.

5.4 Extended Static Analysis

The Extended Static Checker for Java version 2 (ESC/Java2) is a programming tool that attempts to find common run-time errors in JML-annotated Java programs by static analysis of the program code and its formal annotations [5]. Users control the amount and kinds of checking that ESC/Java2 performs by annotating their programs with specially formatted comments called pragmas. ESC/Java2 is used to type check the JML specifications and to check that the Java implementation fulfills these specifications.

ESC/Java2 is used to both type check the JML specifications and to check that the Java implementation fulfills these specifications. When used carefully, ESC/Java2 performs full function verification, as we have done here. This means that, once ESC/Java2 says that a method in Vótáil is correct, then the implementation of that method fulfills its specification for all possible input values on all possible execution paths. This is an extremely strong guarantee, much stronger than even the comprehensive testing that we have done.

This verification is complemented by the aforementioned testing because ESC/Java2 is neither sound nor complete. While we have used its functionality to check that specifications are sound [12] and that we have not ventured into any territory that touches on the soundness and completeness issues inherent in the tool's design and implementation [13], only via rigorous, well-designed testing are we assured that the system is functioning correctly in an actual execution environment.

[7] www.eclemma.org
[8] www.eclipse.org

5.5 Continuous Testing

Every change to the source code committed to the version control repository causes the 7,000+ tests to be run automatically. The test results can be seen at the project website. We are using the open source Hudson Extensible Continuous Integration Server[9] to checkout the latest source code from subversion, run the tests, and summarize the results.

5.6 Beta Release

Vótáil is open source and its test results are public. The beta release is available from the project website[10].

6 Results and Conclusions

We have shown how to specify, implement, validate, and verify, in a traceable fashion, a complex voting scheme such as PR-STV using formal methods. To accomplish such requires a combination of rigorous process, delicate specification techniques, and a novel combination of quality tools. Still, there are a number of problems with our approach and some next steps are critical in realizing trusted elections.

6.1 Near Future Work

Firstly, we seek to identify the optimal minimal number of test cases involving different combinations of ballots to fully test any voting scheme. If there are S seats and C candidates, then how many equivalent election results are possible, where equivalent means either reordering of candidates or magnifying the numbers of ballots in each permutation. This challenge is discussed in a technical report available on the project website.

Secondly, we would like to compare our work with other rigorous implementations of tallying software. Unfortunately, there are no publicly available verified implementations of PR-STV for comparison with Vótáil. The authors have attempted to collaborate with other groups that claim to have work related to us. In all cases to date, it seems that test data and specifications have neither been published nor archived, so we claim that ours is the first publicly available release of formally specified and verified PR-STV software, developed independently of any previous attempt.

[9] http://hudson-ci.org/
[10] http://www.kindsoftware.com/products/opensource/Votail

6.2 Reflections

Voting is but one component of the election process. One of the benefits of electronic voting it it becomes feasible to use more advanced voting schemes that might otherwise take weeks to count by hand.

We claim the first complete formal specification of the Irish PR-STV ballot counting procedure. The requirements are traceable from the legislation, through BON and JML to the Java code. The specifications and source code are publicly available for comment and criticism. There are no other publicly available works of this kind.

PR-STV is one of the most complex voting systems in use today, particularly with regards to formal specification and verification. It is also one of the most complex which can be implemented and understood using paper ballots. This suggests that using a combination of cryptographic schemes with PR-STV for electronic voting will make for an even more difficult verification challenge.

Ireland's Commission on Electronic Voting laid down several recommendations for future use of electronic voting, including the following, some of which were mentioned in section 3:

- clear definition of requirements and specifications,
- robust and formal approach to design and development,
- separation of critical concerns (election management, count rules, vote file, etc.),
- the appropriate use of open source methods,
- publication or public inspection of the source code,
- open public testing of the vote recording software and the vote counting software via an on-line web interface designed to simulate the hardware interfaces of the system, and
- full and formal process of requirements capture and functional specifications for any proposed new system.

This leads us to conclude that the next generation of electronic voting systems in Ireland (if any), will be developed in open source using formal methods, and that each functional module (e.g., ballot counting) will be developed and tested independently. At this time Vótáil is intended to be only a reference implementation that is formally guaranteed to give the correct results for any valid set of ballots. Its use in actual elections is not suggested without further work on verifying inputs and outputs of the system.

Existing verification tools do not yet provide full verification for systems written in Java. Furthermore, as of yet there is no full functional verification tool which has been fully verified in itself. Since we cannot yet achieve full assurance of verification, then we are not yet ready for electronic voting, except for very low-risk elections for example, labor unions or youth/student elections.

Verifiable Elections Verifiable Elections are important. Counting of ballots is only one facet of the entire process, but is a critical component. In small elections, it is feasible to count and recount votes by hand, but the cost of manual

counting and of managing paper ballots in a central facility does not scale for larger populations. Secure storage of ballot boxes in a central facility is often expensive, but PR-STV requires central tallying of ballots in each constituency. Some people might believe that it is less expensive to count votes by hand than to use electronic voting machines. However, if Ireland or other countries like it decide to reduce the size of its parliament, and therefore increase the size of its constituencies, then manual counting starts to become more expensive.

The process of specifying and formalizing the Irish PR-STV count process took almost two man-years of work to complete. As it is the focus of a critical societal system and represents a one-off cost that can be extended and customized to work with other variants of PR-STV, we believe that this is a very reasonable amount of labor.

Critical systems must be designed and constructed with care and consideration. Dependable software engineering techniques are mandatory. If electronic voting is to be used at all, then the software design must be flawless. A typical argument against the use of formal methods is cost and time. This case study shows that the resources necessary are not large and we must balance the cost of formal methods against the cost of rectifying design flaws in electronic voting machines as well as the cost of having to re-run an election. In the end, formal methods look to be faster, cheaper, and better in this particular domain.

7 Acknowledgments

This work is being supported by IT University of Copenhagen, the European project Mobius within the IST 6th Framework and national grants from Science Foundation Ireland including LERO CSET and LERO Graduate School of Software Engineering. This paper reflects only the authors' views and the EU is not liable for any use that may be made of the information contained therein.

References

1. Patrice Chalin, Joseph R. Kiniry, Gary T. Leavens, and Erik Poll. Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In *Proceedings of the International Symposium on Formal Methods for Components and Objects (FMCO)*, volume 4111 of *Lecture Notes in Computer Science*, pages 342–363. Springer–Verlag, 2006.
2. Y. Cheon and G. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 1789–1901, 2002.
3. Yoonsik Cheon and Gary T. Leavens. A simple and practical approach to unit testing: The jml and junit way. In *Proceedings of the European Conference on Object-oriented Programming ECOOP 2002*, volume 2374 of *Lecture Notes in Computer Science*. Springer–Verlag, 2006.
4. D. Cochran. Secure internet voting in Ireland using the Open Source Kiezen op Afstand (KOA) remote voting system. Master's thesis, University College Dublin, March 2006.

5. D.R. Cok and J.R. Kiniry. ESC/Java2: Uniting ESC/Java and JML. In *CASSIS*, volume 3362, pages 108–128. Springer, 2004.
6. L. Coyle, P. Cunnigham, and D. Doyle. Secrecy, accuracy and testing of the chosen electronic voting system: Reliability and accuracy of data inputs and outputs, December 2004.
7. Stéphanie Delaune, Steve Kremer, and Mark Ryan. Verifying properties of electronic voting protocols. In *Proceedings of IAVoSS Workshop On Trustworthy Elections (WOTE)*, 2006.
8. Department of Environment and Local Government, Commission on Electronic Voting. Count requirements and commentary on count rules, 23 June 2000.
9. Department of Environment and Local Government, Commission on Electronic Voting. Count requirements and commentary on count rules, section 16, 23 June 2000.
10. Department of Environment and Local Government, Commission on Electronic Voting. Final Report of Commission on Electronic Voting, July 2006.
11. Fintan Fairmichael. Full verification of the KOA tally system, May 2005.
12. Mikoláš Janota, Radu Grigore, and Michał Moskal. Reachability analysis for annotated code. In *6th International Workshop on the Specification and Verification of Component-Based Systems (SAVCBS)*, Dubrovnik, Croatia, 2007. Workshop at ESEC/FSE 2007.
13. Joseph Kiniry and Alan Morkan. Soundness and completeness warnings in ESC/Java2. In *5th International Workshop on the Specification and Verification of Component-Based Systems (SAVCBS)*, Portland, Oregon, 2006.
14. Joseph Kiniry, Alan Morkan, Dermot Cochran, Fintan Fairmichael, Patrice Chalin, Martijn Oostdijk, and Engelbert Hubbers. The KOA remote voting system: A summary of work to date. In *Proceedings of Trustworthy Global Computing*, 2006.
15. Joseph Kiniry and Daniel Zimmerman. A verification-centric software development process for java. In *Proceedings of the 9th International Conference on Software Quality (QSIC 2009)*, Jeju, Korea, August 2009.
16. Joseph R. Kiniry and Daniel M. Zimmerman. Secret ninja formal methods. In *Proceedings of the Fifteenth International Symposium on Formal Methods (FM)*, volume 5014 of *Lecture Notes in Computer Science*, 2008.
17. G. Leavens, A. Baker, and C. Ruby. JML: A notation for detailed design. *Kluwer International Series in Engineering and Computer Science*, pages 175–188, 1999.
18. Isabel Nunes, Antónia Lopes, and Vasco T. Vasconcelos. Bridging the gap between algebraic specification and object-oriented generic programming. In *Selected Papers from the 9th International Workshop on Runtime Verification (RV)*, volume 5779 of *Lecture Notes in Computer Science*. Springer–Verlag, 2009.
19. R. Sinnott. *Irish voters decide: Voting behaviour in elections and referendums since 1918*. Manchester Univ Press, 1995.
20. V. Teague, K. Ramchen, and L. Naish. Coercion-resistant tallying for STV voting. In *Proceedings of the USENIX/Accurate Electronic Voting Technology Workshop*. USENIX Association Berkeley, CA, USA, 2008.
21. R. van der Meyden. A clausal logic for deontic action specification. In V. Saraswat and K. Ueda, editors, *Proceedings of the International Symposium on Logic Programming*. MIT Press, 1991.
22. Kim Waldén and Jean-Marc Nerson. *Seamless Object-Oriented Software Architecture - Analysis and Design of Reliable Systems*. The Object-Oriented Series. Prentice–Hall, Inc., 1995.

Sawja: Static Analysis Workshop for Java

Laurent Hubert[1], Nicolas Barré[2], Frédéric Besson[2], Delphine Demange[3], Thomas Jensen[2], Vincent Monfort[2], David Pichardie[2], and Tiphaine Turpin[2]

[1] CNRS/IRISA, France
[2] INRIA Rennes - Bretagne Atlantique, France
[3] ENS Cachan - Antenne de Bretagne/IRISA, France

Abstract. Static analysis is a powerful technique for automatic verification of programs but raises major engineering challenges when developing a full-fledged analyzer for a realistic language such as Java. Efficiency and precision of such a tool rely partly on low level components which only depend on the syntactic structure of the language and therefore should not be redesigned for each implementation of a new static analysis. This paper describes the Sawja library: a static analysis framework fully compliant with Java 6 which provides OCaml modules for efficiently manipulating Java bytecode programs. We present the main features of the library, including i) efficient functional data-structures for representing program with implicit sharing and lazy parsing, ii) an intermediate stack-less representation, and iii) fast computation and manipulation of complete programs. We provide experimental evaluations of the different features with respect to time, memory and precision.

Introduction

Static analysis is a powerful technique that enables automatic verification of programs with respect to various properties such as type safety or resource consumption. One particular well-known example of static analysis is given by the Java Bytecode Verifier (BCV), which verifies at loading time that a given Java class (in bytecode form) is type safe. Developing an analysis for a realistic language such as Java is a major engineering task, challenging both the companies that want to build robust commercial tools and the research scientists who want to quickly develop prototypes for demonstrating new ideas. The efficiency and the precision of any static analysis depend on the low-level components which manipulates the class hierarchy, the call graph, the intermediate representation (IR), etc. These components are not specific to one particular analysis, but they are far too often re-implemented in an *ad hoc* fashion, resulting in analyzers whose overall behaviour is sub-optimal (in terms of efficiency or precision). We argue that it is an integral part of automated software verification to address the issue of how to program a static analysis platform that is at the same time efficient, precise and generic, and that can facilitate the subsequent implementation of specific analyzers.

This paper describes the Sawja library — and its sub-component Javalib — which provides OCaml modules for efficiently manipulating Java bytecode programs. The library is developed under the GNU Lesser General Public License and is freely available at `http://sawja.inria.fr/`.

Sawja is implemented in OCaml [17], a strongly typed functional language whose automatic memory management (garbage collector), strong typing and pattern-matching facilities make particularly well suited for implementing program processing tools. In particular, it has been successfully used for programming compilers (*e.g.*, Esterel [24]) and static analyzers (*e.g.*, Astrée [3]).

The main contribution of the Sawja library is to provide, in a unified framework, several features that allow rapid prototyping of efficient static analyses while handling all the subtleties of the Java Virtual Machine (JVM) specification [20]. The main features of Sawja are:

- parsing of **`.class`** files into OCaml structures and unparsing of those structures back into **`.class`** files;
- decompilation of the bytecode into a high-level stack-less IR;
- sharing of complex objects both for memory saving and efficiency purpose (structural equality becomes equivalent to pointer equality and indexation allows fast access to tables indexed by class, field or method signatures, etc.);
- the determination of the set of classes constituting a complete program (using several algorithms, including Rapid Type Analysis (RTA) [1]);
- a careful translation of many common definitions of the JVM specification, *e.g.*, about the class hierarchy, field and method resolution and look-up, and intra- and inter-procedural control flow graphs.

This paper describes the main features of Sawja and their experimental evaluation. Sect. 1 gives an overview of existing libraries for manipulating Java bytecode. Sect. 2 describes the representation of classes, Sect. 3 presents the intermediate representation of Sawja and Sect. 4 presents the parsing of complete programs.

1 Existing Libraries for Manipulating Java Bytecode

Several similar libraries have already been developed so far and some of them provide features similar to some of Sawja's. All of them, except Barista, are written in Java.

The Byte Code Engineering Library[4](BCEL) and ASM[5] are open source Java libraries for generating, transforming and analysing Java bytecode classes. These libraries can be used to manipulate classes at compile-time but also at runtime, *e.g.*, for dynamic class generation and transformation. ASM is particularly optimised for this latter case: it provides a visitor pattern which makes possible local class transformations without even building an intermediate parse-tree.

[4] `http://jakarta.apache.org/bcel/`
[5] `http://asm.ow2.org/`

Those libraries are well adapted to instrument Java classes but lack important features essential for the design of static analyses. For instance, unlike Sawja, neither BCEL nor ASM propose a high-level intermediate representation (IR) of bytecode instructions. Moreover, there is no support for building the class hierarchy and analysing complete programs. The data structures of Javalib and Sawja are also optimized to manipulate large programs.

The Jalapeño Optimizing Compiler [6] which is now part of the Jikes RVM relies on two IR (low and high-level IR) in order to optimize bytecode. The high-level IR is a 3-address code. It is generated using a symbolic evaluation technique described in [30]. The algorithm we use to generate our IR is similar. Our algorithm works on a fixed number of passes on the bytecode while their algorithm is iterative. The Jalapeño high-level IR language provides explicit check instructions for common run-time exceptions (*e.g.*, `null_check`, `bound_check`), so that they can be easily moved or eliminated by optimizations. We use similar explicit checks but to another end: static analyses definitely benefit from them as they ensure expressions are error-free.

Soot [29] is a Java bytecode optimization framework providing three IR: Baf, Jimple and Grimp. Optimizing Java bytecode consists in successively translating bytecode into Baf, Jimple, and Grimp, and then back to bytecode, while performing diverse optimizations on each IR. Baf is a fully typed, stack-based language. Jimple is a typed stack-less 3-address code and Grimp is a stack-less representation with tree expressions, obtained by collapsing Jimple instructions. The IR in Sawja and Soot are very similar but are obtained by different transformation techniques. They are experimentally compared in Sect. 3. Several state-of-the-art control-flow analyses, based on points-to analyses, are available in Soot through Spark [18] and Paddle [19]. Such libraries represent a coding effort of several man-years. To this respect, Sawja is less mature and only proposes simple (but efficient) control-flow analyses.

Wala [15] is a Java library dedicated to static analysis of Java bytecode. The framework is very complete and provides several modules like control flow analyses, slicing analyses, an inter-procedural dataflow solver and a IR in SSA form. Wala also includes a front-end for other languages like Java source and JavaScript. Wala and its IBM predecessor DOMO have been widely used in research prototypes. It is the product of the long experience of IBM in the area. Compared to it, Sawja is a more recent library with less components, especially in terms of static analyses examples. Nevertheless, the results presented in Sect. 4 show that Sawja loads programs faster and uses less memory than Wala. For the moment, no SSA IR is available in Sawja but this could easily be added.

Julia [26] is a generic static analysis tool for Java bytecode based on the theory of abstract interpretation. It favors a particular style of static analysis specified with respect to a denotational fixpoint semantics of Java bytecode. Initially free software, Julia is not available anymore.

Barista [7] is an OCaml library used in the OCaml-Java project. It is designed to load, construct, manipulate and save Java class files. Barista also features a Java API to access the library directly from Java. There are two

representations: a low-level representation, structurally equivalent to the class file format as defined by Sun, and a higher level representation in which the constant pool indices are replaced by the actual data and the flags are replaced by enumerated types. Both representations are less factorized than in JAVALIB and, unlike JAVALIB, BARISTA does not encode the structural constraints into the OCAML structures. Moreover, it is mainly designed to manipulate single classes and does not offer the optimizations required to manipulate sets of classes (lazy parsing, hash-consing, etc).

2 High-level Representation of Classes

SAWJA is built on top of JAVALIB, a JAVA bytecode parser providing basic services for manipulating class files, *i.e.*, an optimised high-level representation of class files, pretty printing and unparsing of class files.[6] JAVALIB handles all aspects of class files, including stackmaps (J2ME and JAVA 6) and JAVA 5 annotation attributes. It is made of three modules: Javalib, JBasics, and JCode[7].

Representing class files constitutes the low-level part of a bytecode manipulation library. Our design choices are driven by a set of principles which are explained below.

Strong typing We use the OCaml type system to explicit as much as possible the structural constraints of the class file format. For example, interfaces are only signaled by a flag in the JAVA class file format and this requires to check several consistency constraints between this flag and the content of the class (interface methods must be abstract, the super-class must be `java.lang.Object`, etc.). Our representation distinguishes classes and interfaces and these constraints are therefore expressed and enforced at the type level. This has two advantages. First, this lets the user concentrate on admissible class files, by reducing the burden of handling illegal cases. Second, for the generation (or transformation) of class files, this provides good support for creating correct class files.

Factorization Strong typing sometimes lacks flexibility and can lead to unwanted code duplication. An example is the use of several, distinct notions of types in class files at different places (JVM types, JAVA types, and JVM array types). We factorize common elements as much as possible, sometimes by a compromise on strong typing, and by relying on specific language features such as polymorphic variants[8]. Fig. 1 describes the hierarchy formed by these types. This factorization principle applies in particular to the representation of op-codes: many

[6] JAVALIB is a sub-component of SAWJA, which, despite being tightly integrated in SAWJA, can also be used as an independent library. It was initiated by Nicolas Cannasse before 2004 but, since 2007, we have largely extended the library. We are the current maintainers of the library.

[7] In the following, we use boxes around JAVALIB and SAWJA module names to make clickable links to the on-line API documentation

[8] Polymorphic variants are a particular notion of enumeration which allows the sharing of constructors between types.

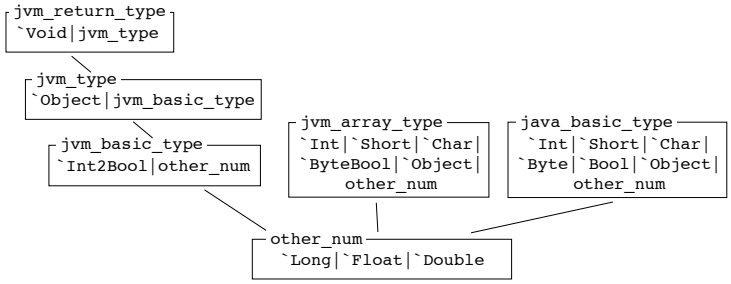


Fig. 1. Hierarchy of JAVA bytecode types. Links represent the subtyping relation enforced by polymorphic variants (for example, the type `jvm_type` is defined by **type** `jvm_type = [ | 'Object | jvm_basic_type ]`).

instructions exist whose name only differ in the JVM type of their operand, and variants exist for particular immediate values (*e.g.*, `iload`, `aload`, `aload_$n$`, etc.). In our representation they are grouped into families with the type given as a parameter (`OpLoad` **of** `jvm_type * int`).

Lazy Parsing To minimise the memory footprint, method bodies are parsed on demand when their code is first accessed. This is almost transparent to the user thanks to the `Lazy` OCAML library but is important when dealing with very large programs. It follows that dead code (or method bodies not needed for a particular analysis) does not cause any time or space penalty.

Hash-consing of the Constant Pool For a JAVA class file, the constant pool is a table which gathers all sorts of data elements appearing in the class, such as Unicode strings, field and method signatures, and primitive values. Using the constant pool indices instead of actual data reduces the class files size. This low-level aspect is abstracted away by the JAVALIB library, but the sharing is retained and actually strengthened by the use of *hash-consing*. Hash-consing [11] is a general technique for ensuring maximal sharing of data-structures by storing all data in a hash table. It ensures unicity in memory of each piece of data and allows to replace structural equality tests by tests on pointers. In JAVALIB, it is used for constant pool items that are likely to occur in several class files, *i.e.*, class names, and field and method signatures. Hash-consing is global: a class name like `java.lang.Object` is therefore shared across all the parsed class files. For JAVALIB, our experience shows that hash-consing is always a winning strategy; it reduces the memory footprint and is almost unnoticeable in terms of running time[9]. We implement a variant which assigns hash-consed values a unique (integer) identifier. It enables optimised algorithms and data-structures. In particular, the JAVALIB API features sets and maps of hash-consed values based on Patricia trees [23], which are a type of prefix tree. Patricia trees are an efficient purely functional data-structure for representing sets and maps of integers, *i.e.*, identifiers of hash-consed values. They exhibit good sharing properties that make them very space efficient. Patricia trees have been proved very

[9] The indexing time is compensated by a reduced stress on the garbage collector.

efficient for implementing flow-sensitive static analyses where sharing between different maps at different program points is crucial. On a very small benchmark computing the transitive closure of a call graph, the indexing makes the computation time four times smaller. Similar data-structures have been used with success in the Astrée analyzer [3].

Visualization SAWJA includes functions to print the content of a class into different formats. A first one is simply raw text, very close to the bytecode format as output by the `javap` command (provided with Sun's JDK).

A second format is compatible with Jasmin [22], a JAVA bytecode assembler. This format can be used to generate incorrect class files (*e.g.*, during a JAVA virtual machine testing), which are difficult to generate with our framework. The idea is then, using a simple text editor, to manually modify the Jasmin files output by SAWJA and then to assemble them with Jasmin, which does not check classes for structural constraints.

Finally, SAWJA provides an HTML output. It allows displaying class files where the method code can be folded and unfolded simply by clicking next to the method name. It also makes it possible to open the declaration of a method by clicking on its signature in a method call, and to know which method a method overrides, or by which methods a method is overridden, etc. User information can also be displayed along with the code, such as the result of a static analysis. From our experience, it allows a faster debugging of static analyses.

3 Intermediate Representation

The JVM is a stack-based virtual machine and the intensive use of the operand stack makes it difficult to adapt standard static analysis techniques that have been first designed for more classic variable-based codes. Hence, several bytecode optimization and analysis tools work on a bytecode *intermediate representation* (IR) that makes analyses simpler [6,29]. Surprisingly, the semantic foundations of these transformations have received little attention. The transformation that is informally presented here has been formally studied and proved semantics-preserving in [10].

3.1 Overview of the IR Language

Fig. 2 gives the bytecode and IR versions of the simple method

```
B f(int x, int y) { return (x==0)?(new B(x/y, new A())):null;}
```

The bytecode version reads as follows : the value of the first argument `x` is pushed on the stack at program point 0. At point 1, depending on whether `x` is zero or not, the control flow jumps to point 4 or 24 (in which case the value **`null`** is returned). At point 4, a new object of class `B` is allocated in the heap and its reference is pushed on top of the operand stack. Its address is then duplicated on the stack at point 7. Note the object is *not initialized* yet. Before the constructor

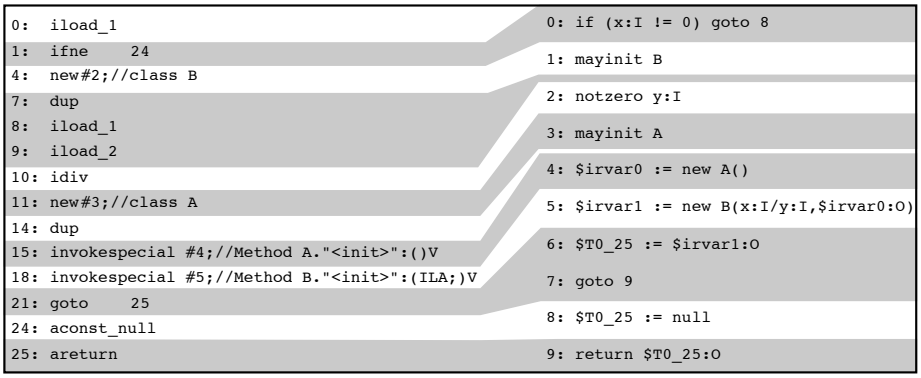


Fig. 2. Example of bytecode (left) (obtained with `javap -c`) and its corresponding IR (right). Colors make explicit the boundaries of related code fragments.

of class `B` is called (at point 18), its arguments must be computed: lines 8 to 10 compute the division of `x` by `y`, lines 11 to 15 construct an object of class `A`. At point 18, the non-virtual method `B` is called, consuming the three top elements of the stack. The remaining reference of the `B` object is left on the top of the stack and represents from now on an *initialized* object.

The right side of Fig. 2 illustrates the main features of the IR language. [10] First, it is *stack-less* and manipulates *structured expressions*, where variables are annotated with *types*. For instance, at point 0, the branching instruction contains the expression `x:I`, where `I` denotes the type of JAVA integers. Another example of recovered structured expression is `x:I/y:I` (at point 5). Second, expressions are *error-free* thanks to explicit checks: the instruction `notzero y:I` at point 2 ensures that evaluating `x:I/y:I` will not raise any error. Explicit checks additionally guarantee that the order in which *exceptions* are raised in the bytecode is preserved in the IR. Next, the object creation process is syntactically simpler in the IR because the two distinct phases of (i) allocation and (ii) constructor call are merged by *folding* them into a single IR instruction (see point 4). In order to simplify the design of static analyses on the IR, we forbid side-effects in expressions. Hence, the nested object creation at source level is decomposed into two assignments (`$irvar0` and `$irvar1` are temporary variables introduced by the transformation). Notice that because of side-effect free expressions, the order in which the `A` and `B` objects are allocated must be reversed. Still, the IR code is able to preserve the *class initialization order* using the dedicated instruction `mayinit` that calls the static class initializer whenever it is required.

[10] For a complete description of the IR language syntax, please refer to the API documentation of the JBir module. A 3-address representation called A3Bir is also available where each expression is of height at most 1.

3.2 IR Generation

The purpose of the SAWJA library is not only static analysis but also lightweight verification [25], *i.e.*, the verification of the result of a static analysis in a single pass over the method code. To this end, our transforming algorithm operates in a fixed number of passes on the bytecode, *i.e.*, without performing fixpoint iteration.

JAVA subroutines (bytecodes `jsr`/`ret`) are inlined. Subroutines have been pointed out by the research community as raising major static analysis difficulties [27]. Our restricted inlining algorithm cannot handle nested subroutines but is sufficient to inline all subroutines from Sun's JAVA 7 JRE.

The IR generation is based on a symbolic execution of the bytecode: each bytecode modifies a stack of symbolic expressions, and potentially gives rise to the generation of IR instructions. For instance, bytecodes at lines 8 and 9 (left part of Fig. 2) respectively push the expressions `x` and `y` on the symbolic stack (and do not generate IR instructions). At point 10, both expressions are consumed to build both the IR explicit check instruction and the expression `x/y` which is then pushed, as a result, on the symbolic stack. The non-iterative nature of our algorithm makes the transformation of jumping instructions non-trivial. Indeed, during the transformation, the output symbolic stack of a given bytecode is used as the entry symbolic stack of all its successors. At a join point, we thus must ensure that the entry symbolic stack is the same regardless of its predecessors. The idea is here to empty the stack at branching points and restore it at join points. More details can be found in [10]. IR expression types are computed using a standard type inference algorithm similar to what is done by the BCV. It only differs in the type domain we used, which is less precise, but does not require iterating. This additionally allows us interleaving expression typing with the IR generation, thus resulting in a gain in efficiency. This lack of precision could be easily filled in using the stackmaps proposed in the JAVA 6 specification.

3.3 Experiments

We validate the SAWJA IR with respect to two criteria. We first evaluate the time efficiency of the IR generation from JAVA bytecode. Then, we show that the generated code contains a reasonable number of local variables. We additionally compare our tool with the SOOT framework. Our benchmark libraries are real-size JAVA code available in `.jar` format. This includes Javacc 4.0 (JAVA Compiler Compiler), JScience 4.3 (a comprehensive JAVA library for the scientific community), the JAVA runtime library 1.5.0_12 and SOOT 2.2.3.

IR Generation Time In order to be usable for lightweight verification, the bytecode transformation must be efficient. This is mainly why we avoid iterative techniques in our algorithm. We compare the transformation time of our tool with the one of SOOT. The results are given in Fig. 3. For each benchmark

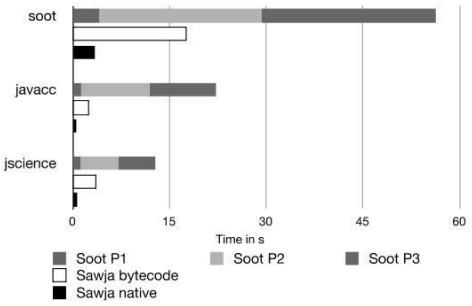


Fig. 3. SAWJA and SOOT IR generation times

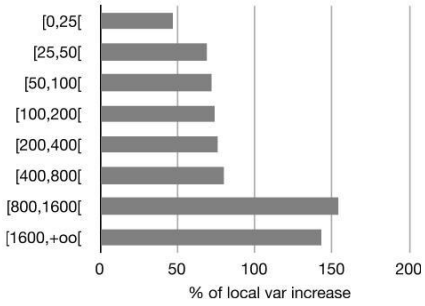


Fig. 4. SAWJA: local variable increase

library[11], we compare our running time for transforming all classes with the running time of SOOT. Here, we choose to generate with SOOT the Grimp representation of classes[12], the closest IR to ours that SOOT provides. Grimp allows expressions with side-effects, hence expressions are somewhat more aggregated than in our IR. However, this does not inverse the trend of results. We rely on the time measures provided by SOOT, from which we only keep three phases: generation of naive Jimple 3-address code (P1), local def/use analysis used to simplify this naive code (P2), and aggregation of expressions to build Grimp syntax (P3). (Other phases, like typing, are not directly relevant.) Unlike JAVA code, OCAML code is usually executed in native form. For the comparison not to be biaised, we compare execution times of both tools in bytecode form and also give the execution time of SAWJA in native form. These experiments show that SAWJA (both in bytecode and native mode) is very competitive with respect to SOOT, in terms of computation efficiency. This is mainly due to the fact that, contrary to SOOT, our algorithm is non-iterative.

Compactness of the Obtained Code Intermediate representations rely on temporary variables in order to remove the use of operand stack and generate side-effect free expressions. The major risk here is an explosion in the number of new variables when transforming large programs.

In practice our tool stays below doubling the number of local variables, except for very large methods (> 800 bytecodes). Fig. 4 presents the percentage of local variable increase induced by our transformation, for each method of our benchmarks, and sorting results according to the method size (indicated by numbers in brackets). The number of new variables stays manageable and we believe it could be further reduced using standard optimization techniques, as those employed by SOOT, but this would require to iterate on each method.

We have made a direct comparison with the SOOT in terms of the local variable increase. Fig. 5 presents two measures. For each method of our benchmarks we count the number N_{SAWJA} of local variables in our IR code and the number

[11] For scale reason, the JAVA runtime library measures are not shown here.
[12] The SOOT transformation is without any optimisation option.

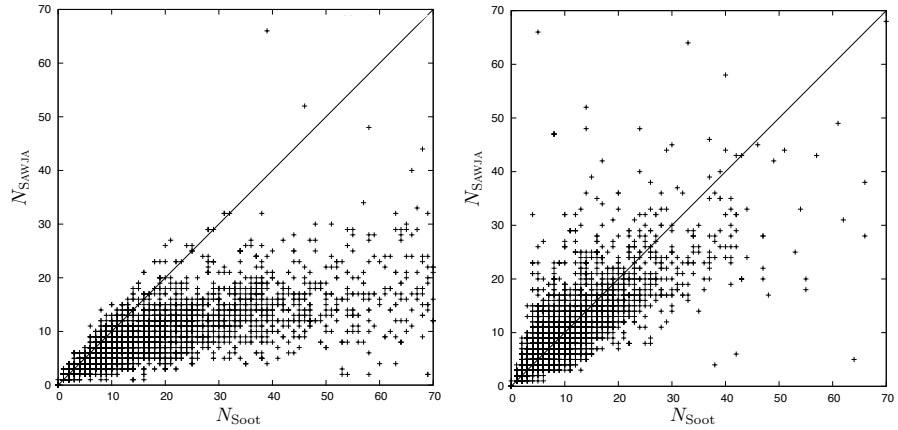


Fig. 5. Local variable increase ratio between SAWJA and SOOT.

N_{Soot} of local variables in the code generated by SOOT. A direct comparison of our IR against Grimp code is difficult because it allows expressions with side-effects, thus reducing the amount of required variables. Hence, in this experiment, the comparison is made between SOOT's 3-address IR (Jimple) and our 3-address IR. For each method we draw a point of coordinate $(N_{Soot}, N_{\text{SAWJA}})$ and see how the points are spread out around the first bisector. For the left diagram, SOOT has been launched with default options. For the right diagram, we added to the SOOT transformation the local packer (`-p jb.lp enabled:true` SOOT option) that reallocates local variables using use/def information. Our transformation competes well, even when SOOT uses this last optimization. We could probably improve this ratio using a similar packing.

4 Complete Programs

Whole program analyses require a model of the global control-flow graph of an entire Java program. For those, SAWJA proposes the notion of *complete programs*. Complete programs are equipped with a high-level API for navigating the control-flow graph and are constructed by a preliminary control-flow analysis.

4.1 API of Complete Programs

SAWJA represents a complete program by a record. The field `classes` maps a class name to a class node in the class hierarchy. The class hierarchy is such that any class referenced in the program is present. The field `parsed_methods` maps a fully qualified method name to the class node declaring the method and the implementation of the method. The field `static_lookup_method` returns the set of target methods of a given field. As it is computed statically, the target methods are an over-approximation.

The API allows navigating the intra-procedural graph of a method taking into account jumps, conditionals and exceptions. Although conceptually simple, field and method resolution and the different method look-up algorithms (corresponding to the instructions `invokespecial`, `invokestatic`, `invokevirtual`, `invokeinterface`) are critical for the soundness of inter-procedural static analyses. In SAWJA, great care has been taken to ensure an implementation fully compliant with the JVM specification.

4.2 Construction of Complete Programs

Computing the exact control-flow graph of a JAVA application is undecidable and computing a precise (over-)approximation of it is still computationally challenging. It is a field of active research (see for instance [19,4]). A complete program is computed by: (1) initializing the set of reachable code to the entry points of the program, (2) computing the new call graph, and (3) if a (new) edge of the call graph points to a new node, adding the node to the set of reachable code and repeating from step (2). The set of code obtained when this iteration stops is an over-approximation of the complete program.

Computing the call graph is done by resolving all reachable method calls. Here, we use the functions provided in the SAWJA API presented in Sect. 4.1. While `invokespecial` and `invokestatic` instructions do not depend on the data of the program, the function used to compute the result of `invokevirtual` and `invokeinterface` need to be given the set of object types on which the virtual method may be called. The analysis needs to have an over-approximation of the types (classes) of the objects that may be referenced by the variable on which the method is invoked.

There exists a rich hierarchy of control-flow analyses trading time for precision [28,12]. SAWJA implements the fastest and most cost-effective control-flow analyses, namely Rapid Type Analysis (RTA) [1], XTA [28] and Class Reachability Analysis (CRA), a variant of Class Hierarchy Analysis [9].

Soundness Our implementation is subject to the usual caveats with respect to reflection and native methods. As these methods are not written in JAVA, their code is not available for analysis and their control-flow graph cannot be safely abstracted. Note that our analyses are always correct for programs that use neither native methods nor reflection. Moreover, to alleviate the problem, our RTA implementation can be parametrised by a user-provided abstraction of native methods specifying the classes it may instantiate and the methods it may call. A better account of reflection would require an inter-procedural string analysis [21] that is currently not implemented.

Implemented Class Analyses

RTA An object is abstracted by its class and all program variables by the single set of the classes that may have been instantiated, *i.e.*, this set abstracts all the

objects accessible in the program. When a virtual call needs to be resolved, this set is taken as an approximation of the set of objects that may be referenced by the variable on which the method is called. This set grows as the set of reachable methods grows.

Sawja's implementation of RTA is highly optimized. While static analyses are often implemented in two steps (a first step in which constraints are built, and a second step for computing a fixpoint), here, the program is unknown at the beginning and constraints are added on-the-fly. For a faster resolution, we cache all reachable virtual method calls, the result of their resolution and intermediate results. When needed, these caches are updated at every computation step. The cached results of method resolutions can then be reused afterwards, when analyzing the program.

XTA As in RTA, an object is abstracted by its class and to every method and field is attached a set of classes representing the set of object that may be accessible from the method or field. An object is accessible from a method if: (i) it is accessible from its caller and it is of a sub-type of a parameter, or (ii) it is accessible from a static field which is read by the method, (iii) it is accessible from an instance field which is read by the method and there an object of a sub-type of the class in which the instance fields is declared is already accessible, or (iv) it is returned by a method which may be called from the current method.

To facilitate the implementation, we built this analysis on top of another analysis to refine a previously computed complete program. This allows us using the aforementioned standard technique (build then solve constraints). For the implementation, we need to represent many class sets. As classes are indexed, these sets can be implemented as sets of integers. We need to compute fast union and intersection of sets and we rarely look for a class in a set. For those reasons, the implementation of sets available in the standard library in OCaml, based on balanced trees, was not well adapted. Instead we used a purely functional set representation based on Patricia trees [23], and another based on BDDs [5] (using the external library BuDDy available at `http://buddy.sourceforge.net`).

CRA This algorithm computes the complete program without actually computing the call graph or resolving methods: it considers a class as accessible if it is referenced in another class of the program, and considers all methods in reachable classes as also reachable. When a class references another class, the first one contains in its constant pool the name of the later one. Combining the lazy parsing of our library with the use of the constant pool allows quickly returning a complete program without even parsing the content of the methods. When an actual resolution of method, or a call graph is needed, the Class Hierarchy Analysis (CHA) [9] is used. Although parts of the program returned by CRA will be parsed during the overlying analysis, dead code will never by parsed.

		Soot	Jess	Jml	VNC	ESC/Java	JDTCore	Javacc	JLex
C	CRA	5,198	5,576	2,943	5,192	2,656	2,455	2,172	2,131
	RTA	4,116	2,222	1,641	1,736	1,388	1,163	792	752
M	CRA	49,810	47,122	26,906	44,678	23,229	23,579	19,389	18,485
	W-RTA	32,652	4,303	17,740	?	9,560	7,378	3,247	1,419
	RTA	32,800	12,561	11,697	9,218	8,305	9,137	4,029	3,157
	XTA	14,251	10,043	9,408	6,534	7,039	8,186	3,250	2,392
	W-0CFA	37,768	9,927	15,414	?	9,088	6,830	3,009	1,186
E	CRA	2,159,590	799,081	418,951	694,451	354,234	347,388	258,674	244,071
	W-RTA	2,788,533	78,444	614,216	?	279,232	146,119	34,192	13,256
	RTA	1,400,958	141,910	149,209	79,029	101,257	114,454	35,727	23,209
	XTA	297,754	94,189	103,126	48,817	74,007	86,794	26,844	15,456
	W-0CFA	856,180	183,191	187,177	?	87,163	77,875	21,475	4,360
T	CRA	8	8	4	7	4	5	4	4
	W-RTA	74	7	23	?	12	12	7	5
	RTA	13	4	4	3	3	4	2	2
	XTA	187	18	16	11	10	14	5	4
	W-0CFA	2,303	209	40	?	27	26	16	7
S	CRA	87	83	51	80	45	47	36	35
	W-RTA	248	44	128	?	84	101	42	8
	RTA	132	60	54	51	43	52	26	20
	XTA	810	198	184	153	147	157	112	107
	W-0CFA	708	238	215	?	132	134	125	26

Table 1. Comparison of algorithms generating a program call graph (with Sawja and Wala): the different algorithms of Sawja (CRA,RTA and XTA) are compared to Wala (W-RTA and W-0CFA) with respect to the number of loaded classes (C), reachable methods (M) and number of edges (E) in the call graph, their execution time (T) in seconds and memory used (S) in megabytes. Question marks (?) indicate clearly invalid results.

Experimental Evaluation We evaluate the precision and performances of the class analyses implemented in Sawja on several pieces of Java software[13] and present our results in Table 1. We compared the precision of the 3 algorithms used to compute complete programs (CRA, RTA and XTA) with respect to the number of reachable methods in the call graph and its number of edges. We also give the number of classes loaded by CRA and RTA. We provide some results obtained with Wala (version r3767 from the repository). Although precision is hard to compare[14], it indicates that, on average, Sawja uses half the memory and time used by Wala per reachable method with RTA.

Conclusion

We have presented the Sawja library, the first OCaml library providing state-of-the-art components for writing Java static analyzers in OCaml.

[13] Soot (2.3.0), Jess (7.1p1), JML (5.5), TightVNC Java Viewer (1.3.9), ESC/Java (2.0b0), Eclipse JDT Core (3.3.0), Javacc (4.0) and JLex (1.2.6).

[14] Because both tools are unsound, a greater number of method in the call graph either mean there is a precision loss or that native methods are better handled.

The library represents an effort of 1.5 man-year and approximately 22000 lines of OCaml (including comments) of which 4500 are for the interfaces. Many design choices are based on our earlier work with the NIT analyzer [13], a quite efficient tool, able to analyze a complete program of more than 30000 methods to infer nullness annotations for fields, method signatures and local variables in less than 2 minutes, while proving the safety of 80% of dereferences. Using our experience from the NIT development, we designed Sawja as a generic framework to allow every new static analysis prototype to share the same efficient components as NIT. Indeed, Sawja has already been used in two implementations for the ANSSI (The French Network and Information Security Agency) [16,14], Nit is being ported to the current version of Sawja, and other small analyses (liveness, interval analyses, etc.) are available on Sawja's web site.

Several extensions are planned for the library. Displaying static analysis results is a first challenge that we would like to tackle. We would like to facilitate the transfer of annotations from Java source to Java bytecode and then to IR, and the transfer of analysis results in the opposite direction. We already provide HTML outputs but ideally the result at source level would be integrated in an IDE such as Eclipse. This manipulation has been already experimented in one of our earlier work for the NIT static analyzer and we plan to integrate it as a new generic Sawja component. To ensure correctness, we would like to replace some components of Sawja by certified extracted code from Coq [8] formalizations. A challenging candidate would be the IR generation that relies on optimized algorithms to transform in at most three passes each bytecode method. We would build such a work on top of the Bicolano [2] JVM formalization that has been developed by some of the authors during the European Mobius project.

References

1. D. F. Bacon and P. F. Sweeney. Fast static analysis of C++ virtual function calls. In *Proc. of OOPSLA'96*, pages 324–341, 1996.
2. Bicolano - web home. `http://mobius.inria.fr/bicolano`.
3. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Proc. of PLDI'03*, pages 196–207, San Diego, California, USA, June 7–14 2003. ACM Press.
4. M. Bravenboer and Y. Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. *SIGPLAN Not.*, 44(10):243–262, 2009.
5. R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Survey*, 24(3):293–318, 1992.
6. M G. Burke, J. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M J. Serrano, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño dynamic optimizing compiler for java. In *Proc. of JAVA'99*, pages 129–141. ACM, 1999.
7. Xavier Clerc. Barista. `http://barista.x9c.fr/`.
8. The Coq Proof Assistant. `http://coq.inria.fr/`.
9. J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proc. of ECOOP'95*, volume 952 of *LNCS*, pages 77–101. Springer, August 1995.

10. D. Demange, T. Jensen, and D. Pichardie. A provably correct stackless intermediate representation for Java bytecode. Research Report 7021, INRIA, 2009. http://www.irisa.fr/celtique/ext/bir/rr7021.pdf.
11. A. P. Ershov. On programming of arithmetic operations. *Commun. ACM*, 1(8):3–6, 1958.
12. D. Grove and C. Chambers. A framework for call graph construction algorithms. *Toplas*, 23(6):685–746, 2001.
13. L. Hubert. A Non-Null annotation inferencer for Java bytecode. In *Proc. of PASTE'08*, pages 36–42. ACM, November 2008.
14. Laurent Hubert, Thomas Jensen, and Vincent. Enforcing secure object initialization in java. Technical report, CNRS, INRIA, April 2010. Submitted to ESORICS 2010.
15. IBM. The T.J. Watson Libraries for Analysis (Wala). http://wala.sourceforge.net.
16. T. Jensen and D. Pichardie. Secure the clones: Static enforcement of policies for secure object copying. Technical report, INRIA, June 2010. Presented at OWASP 2010.
17. X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. *The Objective Caml system*. Inria, May 2007. caml.inria.fr/ocaml/.
18. O. Lhoták and L. Hendren. Scaling Java points-to analysis using Spark. In *Proc. of CC*, volume 2622 of *LNCS*, pages 153–169. Springer, 2003.
19. O. Lhoták and L. Hendren. Evaluating the benefits of context-sensitive points-to analysis using a bdd-based implementation. *ACM Trans. Softw. Eng. Methodol.*, 18(1), 2008.
20. T. Lindholm and F. Yellin. *The Java™ Virtual Machine Specification, Second Edition*. Prentice Hall PTR, 1999.
21. V. Benjamin Livshits, John Whaley, and Monica S. Lam. Reflection analysis for java. In *Proc. of APLAS*, pages 139–160. Springer, 2005.
22. J. Meyer and T. Downing. *Java Virtual Machine*. O'Reilly Associates, 1997. http://jasmin.sourceforge.net.
23. D. R. Morrison. PATRICIA — Practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4), 1968.
24. B. Pagano, O. Andrieu, T. Moniot, B. Canou, E. Chailloux, P. Wang, P. Manoury, and J.L. Colaço. Experience report: using Objective Caml to develop safety-critical embedded tools in a certification framework. In *Proc. of ICFP*, pages 215–220. ACM, 2009.
25. E. Rose. Lightweight bytecode verification. *J. Autom. Reason.*, 31(3-4):303–334, 2003.
26. F. Spoto. Julia: A Generic Static Analyser for the Java Bytecode. In *Proc. of the Workshop FTfJP*, 2005.
27. R. Stata and M. Abadi. A type system for java bytecode subroutines. In *Proc of POPL,98*, pages 149–160. ACM Press, 1998.
28. F. Tip and J. Palsberg. Scalable propagation-based call graph construction algorithms. In *Proc. of OOPSLA'00*, pages 281–293. ACM Press, October 2000.
29. R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - A Java bytecode optimization framework. In *Proc. of CASCON*, 1999.
30. J. Whaley. Dynamic optimization through the use of automatic runtime specialization. Master's thesis, Massachusetts Institute of Technology, May 1999.

State-based Object Models Are More Abstract Than Trace-based Models

Towards a Unified Specification Framework

Ilham W. Kurnia, Arnd Poetzsch-Heffter, Yannick Welsch*

University of Kaiserslautern, Germany
ilham,poetzsch,welsch@cs.uni-kl.de

Abstract. The literature distinguishes between trace-based and state-based specification techniques for object-oriented components. Trace-based specifications describe behavior in terms of the message histories of components, while state-based techniques explain component behavior in terms of states. The latter define how the state is changed by method calls and what is returned as a result. The state space is either abstract or concrete. Abstract states are used to model the behavior without referring to the implementation. Concrete states are expressed by the underlying implementation. State-based specifications are usually described in terms of pre- and postconditions of methods.
In this paper, we investigate the relationship between trace-based specifications and specifications based on abstract states for sequential, object-based components. We first generalize state-based techniques so that they can handle callbacks. Then, we develop formal models for trace-based and state-based specifications and show that every trace-based model can be canonically represented as a state-based model. Adapting notions from process simulation, we define an abstraction relation between two state-based models allowing their comparison. In particular, state-based models are more abstract than trace-based models. We also show that there exist most abstract models. The developed framework is illustrated by a subject component of the Subject-Observer Pattern.

1 Introduction

Objects or, more generally, object-based components communicate with their environment via messages. Usually, the reaction to an incoming message depends on the state of the object or component. To avoid uncontrolled access and to achieve implementation-independency, it is an accepted principle to encapsulate the *concrete state* of the implementation. In particular, the concrete state should not be exposed in specifications of classes and components (see, e.g., [1, §1.3]). Two different kinds of implementation-independent specification techniques have been investigated in the literature. In so-called *model-* or *abstract state-based* specifications, behavior is explained based on a model or space of abstract states

* This work is partly funded by the EU project FP7-231620 HATS: Highly Adaptable and Trustworthy Software using Formal Methods

(see, e.g., [2]). The specification expresses the result values and changes of the abstract state in reaction to an incoming message. In the following, we will simply call these specifications *state-based*. In *trace-based* specifications, behavior is explained with respect to the history of messages an object or component has seen. In both cases, we say that a specification describes a *model* of the specified component. Models can be considered as the semantics of specifications. They will be formalized as transitions systems.

State-based specifications are closer to the implementation, can directly be complemented with invariants – which is important for verifying implementations –, and are supported by a set of well-developed specification constructs going far beyond the basic pre- and postconditions (see, e.g., [3,4]). On the other hand, trace-based specifications have a natural link to semantics (see [5]), avoid the design of an abstract state space, and can more easily deal with callbacks and concurrency (see, e.g., [6, §5]). Furthermore, invariants may also be specified to restrict, e.g., the structure of the allowed traces.

The long-term goal of our research is to work out the relationship between state- and trace-based specifications to combine the best of the two techniques. In this paper we investigate, as an initial step, specifications of sequential components with possible callback behavior. We consider specifications of object-based components where a component description consists of one or more classes. A (runtime) component is created by instantiating a class. Internally, the component can create further objects and expose these objects to the environment. To keep the presentation focused, we do not discuss concurrency and inheritance. However, we believe that our results can be generalized to such settings.

Contributions. To handle callback scenarios, we first generalize state-based specifications. In addition to JML-like pre- and postconditions of methods calls [4], we also allow to express what a component ensures when a call leaves the component and what it requires when the call is completed (returns). In this paper, this generalization is only to ease comparison between the different models. We illustrate the approach for a simple version of a `Subject` component following the Subject-Observer Pattern (SOP) [7].

The central contribution of the paper is to relate trace-based and state-based models in a formal way. In particular, we show that every trace-based model has an equivalent state-based model. We define an abstraction relation between two models which allows to compare two models. As results from the process simulation theory, we derive that there exist most abstract models. Knowing that a model is most abstract guarantees that it does not contain redundant states. Thus, the knowledge can be used as a guidance to remove redundancy from specifications. As an example, we show that the state-based model of the presented subject component example is most abstract.

Paper structure. Section 2 presents the specifications of the `Subject` component of the SOP in trace- and state-based manners. Sections 3 and 4 formally define our trace- and state-based models, respectively, and show how a trace-based model can be represented as a state-based model. Section 5 defines the

```
interface Observer {
  void notify(State s);
}

class Subject {
  Observer o1, o2;

  Subject(Observer o1, Observer o2) {
    this.o1 = o1; this.o2 = o2;
  }

  void update(State s) {
    o1.notify(s);
    o2.notify(s);
  }
}
```

Fig. 1. A `Subject` implementation

abstraction and states that state-based models are abstractions of trace-based models. We then relate this work to others and give some directions regarding future work. Proofs of the lemmas are available in the full version of this paper[1].

2 Motivating Example

In Fig. 1, we give an implementation of the SOP. For simplicity, the `Subject` class stores exactly two observers. Each time the subject is updated with a new state, this state is propagated to the observers in a fixed order (i.e., `o1` then `o2`). In the presence of callbacks, there can be a sequence of notifications to `o1` before `o2` is notified. For example, the observer `o1`, upon being notified, may trigger a new update of the subject. The implementation does not guarantee that the second observer receives the states in the same order as the first observer.

The behavior of the SOP can be described using state- or trace-based specifications. For both kind of specification techniques, we need to address what the input and output of an object-oriented component are.

Input/Output. In general, input/output occurs at the places where the control flow enters or leaves the component. In our example, control flow can enter the component when the constructor or the `update` method is called by an object (outside of the component/from the *environment*). Control flow may leave the component when the constructor or the `update` method returns. One must also note that control flow leaves the component when the `notify` method is

[1] `http://softech.informatik.uni-kl.de/Homepage/PublikationsDetail?id=150`

called. We thus characterize the input/output of our `Subject` component by the following set of messages *Msg*.

$$\begin{aligned} Msg = \; & \{\rightarrow sbj.\texttt{Subject}(o_1, o_2), \leftarrow sbj.\texttt{Subject}()\} \\ & \cup \{\rightarrow sbj.\texttt{update}(s), \leftarrow sbj.\texttt{update}()\} \\ & \cup \{\rightarrow o.\texttt{notify}(s), \leftarrow o.\texttt{notify}()\} \end{aligned}$$

The first subset represents the messages dealing with the construction of the `Subject` instance *sbj*. Invocation of the constructor is represented by the message $\rightarrow$ $sbj.\texttt{Subject}(o_1, o_2)$ where o_1 and o_2 are the parameters passed to the invocation. Returning from the constructor is represented by the message $\leftarrow$ $sbj.\texttt{Subject}()$. Note that each invocation message has a matching completion message in *Msg*. We assume that the instance variables of the `Subject` class cannot be accessed directly by other objects, i.e., that there is no interaction with a `Subject` object other than through its methods. We can thus solely rely on the previously defined messages to describe the input/output behavior of our component. Using these messages, we give both a state-based and a trace-based specification of the SOP. We assume our specifications to be deterministic in the following way. For each input to the component, there is exactly one output which is specified.

State-based specification. A state-based specification is realized using some *abstract* states by describing how a state changes when an event occurs. Our specification of the `Subject` component (Fig. 2) provides an example of a state-based specification in a pre-/postcondition style. In our example, the state consists of a subject, its observers and a stack which stores, in last-in-first-out manner, the states that are available during an update process. The stack is necessary to manage multiple states in the presence of callbacks. We then define the state transitions. A state-based specification consists of a set of specification cases of the following form

in *MsgPattern* **out** *MsgPattern* **requires** *pre*; **ensures** *post*;

A transition is specified by a quadruple where the first entry describes the (incoming) message pattern upon which the transition might be triggered. The second entry then describes the outgoing message (i.e., the response) from the component. The third entry forms a precondition over the values of the incoming message pattern and the state upon which the transition might be selected. The fourth entry represents the postcondition and ranges over the values in the pre-state (denoted by *old*(...)), the post-state and the values of the in- and outgoing message. In short, if an incoming message which enters the component fits the incoming message pattern and the precondition *pre* holds, then the component responds by producing an outgoing message and changing its state such that the postcondition *post* holds. To make our specifications short and concise, we assume the part of the state which is not mentioned in the postcondition to retain the same value it had prior to receiving the incoming message.

```
state spec Subject {
  Subject sbj;
  Observer o1, o2;
  Stack<State> st;

  in → sbj.Subject(o1', o2') out ← sbj.Subject()
    requires o1' ≠ o2' ∧ o1' ≠ null ∧ o2' ≠ null;
    ensures o1 = o1' ∧ o2 = o2' ∧ st = Stack.Empty();

  in → sbj.update(s) out → o1.notify(s)
    ensures st = old(st).push(s);

  in ← o.notify() out → o2.notify(s)
    requires o = o1;
    ensures s = old(st).top();

  in ← o.notify() out ← sbj.update()
    requires o = o2;
    ensures st = old(st).pop();
}
```

Fig. 2. A state-based specification of the `Subject` class

For the concrete example given above, there are four *cases* which need specifications. Given an instance creation request $\rightarrow$ *sbj*.`Subject`(o_1, o_2) with the precondition of two distinct and non-null observer parameters, the `Subject` returns the constructor completion message $\leftarrow$ *sbj*.`Subject`() and the new state refers to the two given observers and an empty stack. If the component receives an `update` message, then o_1 is notified while the state s is pushed onto the stack. As the control flow leaves the component when the `notify` method is called and control flow enters the component again when the `notify` method returns, we also have to take returning `notify` messages into account. These might happen at two different places; after the first or the second observer has been notified. In our specification, when a notification has finished, we check which observer has been involved in this notification. If it is o_1, then the `Subject` proceeds to notify o_2 using the state which is on top of the stack *st*. Otherwise, we conclude that the `update` invocation finishes and pop the stack.

A contract-based specification like JML [4] also employs a pre-/postcondition (state-based) specification style. However, as this is based on methods and not message pattern, we can only consider transition descriptions where the incoming message pattern is an invocation and the outgoing message is the corresponding return. We see our specification style as a generalization of the JML pre-/postcondition style as we can additionally handle callbacks.

Trace-based specification. Another way to describe the SOP behavior is by using a trace-based specification. Here, the only information used by the spec-

```
trace spec Subject {
  in → sbj.Subject(o'1, o'2) out ← sbj.Subject()
    requires o'1 ≠ o'2 ∧ o'1 ≠ null ∧ o'2 ≠ null;

  in → sbj.update(s) out → o1.notify(s)
    requires sbj = sbj(h);
    ensures o1 = obs1(h);

  in ← o1.notify() out → o2.notify(s)
    requires o1 = obs1(h);
    ensures o2 = obs2(h) ∧ s = getS(h);

  in ← o2.notify() out ← sbj.update()
    requires o2 = obs2(h);
    ensures sbj = sbj(h);
}
```

Fig. 3. A trace-based specification of the `Subject` class

ification is the history of messages that have crossed the boundary of the component. This notion of *communication history* [8] is modeled in our specification as a list of incoming and outgoing messages. Similar to before, the specification is described using a pre- and postcondition style, with the difference that we do not use an auxiliary abstract state space but only relate message values to the trace history.[2] Specification cases are of the following form

in *MsgPattern* **out** *MsgPattern* **requires** *pre*; **ensures** *post*; ,

where both pre- and postcondition can refer to the history h (also called *history variable* in [9]). The history h contains all messages seen so far, except the ones which were currently matched against the **in** and **out** message pattern.

The specification for the constructor as seen in Fig. 3 is similar to that from the state-based specification. In contrast to the state-based model, upon receiving an `update`, we must go back in the history h to the constructor message to obtain the subject and the first observer that needs to be notified (represented by the extractor functions `sbj` and `obs1` which are not formally stated here).

The trickiest part to specify is when a notification has happened. In the case where the incoming return message comes from o_1, we need to notify o_2 using the proper state. The problem is that since a callback is allowed and we do not store the states in a stack, we need to obtain this information by emulating the stack effect of each update, which is done by the `getS` function. It searches for the update message containing the state which is relevant to the "stack frame" of the current notification. If the incoming return message comes from o_2, the component returns with an `update` completion message.

[2] Note that this is only one possible way to describe the set of admissible traces.

Comparison. Both state- and trace-based specification techniques have advantages and disadvantages. On one hand, it is sometimes very difficult in the trace-based approach to extract the necessary information from the trace history (e.g., defining `getS`). The state-based approach allows one to define an abstract state which contains all the relevant information one needs to specify the behavior. On the other hand, state changes must then be described. Some protocol properties are easier to specify in a trace-based way, e.g., using regular expressions.

3 Trace-based Model

A trace of a component is a sequence of events which consist of incoming and outgoing messages. It describes how the component responds given a specific instance of an environment which uses the component. A trace-based model can be seen as a collection of such traces which restricts the behavior of the component. We adopt the formalization of messages and traces from [9,10] to define a trace-based model of a sequential deterministic object-based component.

Let *Obj* be a set of objects, *Mtd* a set of methods with unique names, and *Dir* a two-element set $\{\rightarrow, \leftarrow\}$ representing method *invocation* (call) and *completion* (return), respectively. Furthermore, let *Value* be a set of values which encompasses all possible parameter values in actual method calls and return values. Then, with $\overline{v}$ being a shorthand for a possibly empty list of values $v_1, v_2, \ldots$, the set of messages can be defined as follows.

Definition 1 (Message). *The set of* messages *Msg (whose instances are denoted by μ) is a subset of $Obj \times Mtd \times List\langle Value\rangle \times Dir$. A tuple $\mu = \langle o, m, \overline{v}, d\rangle$ is a message if the callee o supports the method m.*

Instead of representing messages μ in the tuple format, we depict them graphically: $\rightarrow o.m(\overline{v})$ or $\leftarrow o.m(\overline{v})$. Invocation messages $\rightarrow o.m(\overline{v})$ are grouped into $Msg^{\rightarrow}$ and completion messages into $Msg^{\leftarrow}$, forming a partitioning of *Msg*. We also define extra functions *callee*, *method*, *value*, and *dir* to extract the callee object, method, value and direction elements from a message, respectively. The function *header* extracts the callee and method of a message.

We define a component as a collection of objects $O \subseteq Obj$ (called *component objects*). All other objects (called *environment objects*) are considered as part of the environment $O_{env} = Obj \backslash O$.

Based on this partitioning, we can categorize the messages into incoming and outgoing messages from the perspective of the component. An *incoming message* is either an invocation message whose callee is a component object, or a completion message whose callee is an environment object. An *outgoing message* is either an invocation message whose callee is an environment object, or a completion message whose callee is a component object. As we are interested in modeling the observable behavior of the component, we only deal with traces composed of alternating incoming and outgoing messages.

Definition 2 (Component Trace). *Given a set of messages Msg and a component O, a* component trace *t is a (possibly empty or infinite) sequence of pairs*

of incoming and outgoing messages $(\mu_1, \mu_2), (\mu_3, \mu_4), \ldots$*, where odd-indexed messages are incoming and even-indexed messages outgoing.*[3]

Note that in the definition above, we assume that the component may not diverge. To include divergence, we could extend *Msg* to include a special message symbol (e.g., $\perp$) to indicate this situation.

In the sequential setting, a component trace must follow the call stack property, where method completions must appear in reverse order of the corresponding method invocations. This is captured by the following definition of well-formed component traces.

Definition 3 (Well-formed Component Trace). *A non-empty component trace* $t = \mu_1, \mu_2, \ldots$ *is* well-formed *with respect to a component* O *and the environment* $O_{env} = Obj \backslash O$*, iff for each completion message there exists a prior invocation message such that the call stack property (predicate match) holds:* $\forall \mu_j \in Msg^{\leftarrow} \bullet \exists k < j \bullet match(k, j)$*, where*

$$\begin{aligned} match(a,b) &\stackrel{def}{=} \mu_a \in Msg^{\rightarrow} \wedge \mu_b \in Msg^{\leftarrow} \wedge \\ &\quad header(\mu_a) = header(\mu_b) \wedge split(a+1, b-1), \text{ and} \\ split(a,b) &\stackrel{def}{=} a > b \vee match(a,b) \vee \\ &\quad \exists\ a < c < b-1 \bullet split(a,c) \wedge split(c+1,b) \end{aligned}$$

An empty trace is well-formed.

The well-formedness condition above states that every completion message needs to have a matching invocation message. The proper matching has to be selected such that no two completion messages are matched to a single invocation message, which is captured by the *match* predicate. The *match* and *split* predicates are mutually recursive, where *match* matches the invocation and completion messages at the two ends of the subtrace (i.e., μ_a and μ_b), and *split* partitions the rest of the subtrace such that for each partition, we can form the matching. Other sequential properties[4] are not described as they complicate the definitions and proofs without adding substantial insight.

We define $Traces(O)$ as the set of well-formed traces of the component O. As we do not specify the behavior of the environment, it may decide to terminate at any moment. Therefore, we require the set $Traces(O)$ to be prefix-closed, i.e., for each trace $t \in Traces(O)$, all its prefixes must also be in $Traces(O)$. A trace-based model is then simply defined using the set of traces.

Definition 4 (Trace-based Model). *Given a set of messages Msg and a component* O*, a* trace-based model *is the set of well-formed traces:*

$\mathcal{T}(Msg, O) = Traces(O)$.

[3] We drop the brackets surrounding the pairs whenever it is clear from the context.

[4] Examples of other sequential properties: all traces begin with the construction of a component; a component cannot send a message to an object from the environment before that object has been introduced to the component, etc.

Because in this paper the set of messages *Msg* and objects of the component O is clear from the context, we simply write $\mathcal{T}$ and *Traces* for the trace-based model and its set of well-formed traces, respectively.

Since we want a model for a deterministic component, we also require that the component acts as a function given some trace prefix and incoming message to produce exactly one outgoing message. Stated formally,

$$\forall t, t', t'' \in \mathit{Traces} \bullet t' = t, \mu_{2n-1}, \mu_{2n} \wedge t'' = t, \mu'_{2n-1}, \mu'_{2n} \wedge \mu_{2n-1} = \mu'_{2n-1} \implies \mu_{2n} = \mu'_{2n} \ .$$

Example (Trace-based model for the `Subject` *component).* The trace-based model $\mathcal{T}_{subj}$ can be defined using the set of well-formed traces Traces_{subj} generated by the specification in Fig. 3. The set of traces Traces_{subj} is inductively constructed. The empty trace is element of the set. Then, for any element t of the set, the trace $t' = t, \mu_1, \mu_2$ is also element of the set, if it satisfies the following conditions. There must be a specification case with **in** and **out** message pattern that are matched by μ_1 and μ_2 and where both pre- and postconditions are satisfied. Furthermore, the trace t' must be well-formed (see Def. 3).

4 State-based Model

Another way to specify the behavior of a component is to determine how the component would act given a request from the environment by looking at the state of the component, updating the state, and forming a response based on it. This is captured by the well-known notion of transition system. In this section, we describe state-based models, defined as transition systems, and how we can represent the trace-based models as state-based models.

Definition 5 (State-based Model). *Given a set of messages Msg and a component O, a* state-based model $\mathcal{M}(\mathit{Msg}, O)$ *is a triple $\langle S, \Theta, s_0 \rangle$ where S is a set of states, $s_0 \in S$ is the initial state and $\Theta \subseteq S \times \mathit{Msg} \times \mathit{Msg} \times S$ is the transition relation, where the first message represents an incoming message and the second one an outgoing message.*

As with trace-based models, we drop the set of messages *Msg* and objects of the component O from the argument of $\mathcal{M}$. We also represent a transition between two states s, s' graphically as $s \xrightarrow{\mu_a, \mu_b} s'$. A (finite) component trace $t = (\mu_1, \mu_2), \ldots$ is induced by $\mathcal{M}$ if there exists a sequence of states $s_0, s_1, \ldots$ such that $\forall i > 0 \bullet s_{i-1} \xrightarrow{\mu_{2i-1}, \mu_{2i}} s_i$. We require our state-based models to be *concise*. A model is concise if each state is reachable by some sequence of transitions from the initial state.

Example (State-based model of the `Subject` *component).* The state-based model $\mathcal{M}_{subj} = \langle S, \Theta, s_0 \rangle$ is a state-based model of the `Subject` component specified in Fig. 2. It has $S \subseteq$ `Subject` $\times$ `Observer` $\times$ `Observer` $\times$ `Stack<State>` as set of states and $s_0 =$ (**`null,null,null,null`**) as initial state. The transition

relation Θ can be derived from the state-based specification in a similar way as how the trace-based model was derived from the trace-based specification. Note however that no well-formedness property must yet hold.

The lemma below formulates trace-based models in terms of state-based models.

Lemma 1. *Every trace-based model $\mathcal{T}$ can be canonically represented as a state-based model $\mathcal{M} = \langle S, \Theta, s_0 \rangle$.*

Proof (sketch). By construction. Let S be a set of finite traces from *Traces* and s_0 be the empty trace ϵ. The transition relation Θ can be built in the following way. We start from an empty relation. Now take any two elements t, t' of *Traces* such that $t' = t, \mu_a, \mu_b$. Then, $t \xrightarrow{\mu_a, \mu_b} t'$ is added to Θ. This construction is similar to how one would build a trie (re*trie*val tree [11]) from a set of strings.

The usual notion of determinism for transition systems is that for any given state s and a transition label (μ_a, μ_b) there is at most one state s' such that $s \xrightarrow{\mu_a, \mu_b} s'$. To include the desired notion of determinism stated in the previous section, we need to strengthen this notion by also requiring that for any state s and an incoming message μ_a, there is at most one outgoing message μ_b and one next state s' such that $s \xrightarrow{\mu_a, \mu_b} s'$. It is not enough to use the determinism of the trace-based view, since the resulting state-based model may be non-deterministic in traditional sense while inducing the same set of traces.

Definition 5 alone is not strong enough to ensure that the resulting component traces are well-formed. There are, for example, state-based models which may induce a trace which breaks the call stack requirement. To enforce the well-formedness requirement, one can build a specific model for each requirement (called *restrictor models*), and then take the synchronous product between the restrictor models and the original state-based model.

Definition 6 (Synchronous Product). *Given two state-based models $\mathcal{M}_i = \langle S_i, \Theta_i, s_{0,i} \rangle, i = 1, 2$, their* synchronous product *$\mathcal{M}' = \mathcal{M}_1 \otimes \mathcal{M}_2$ is $\langle S', \Theta', s'_0 \rangle$ where $S' \subseteq S_1 \times S_2$, $s'_0 = (s_{0,1}, s_{0,2})$, and $\Theta' \subseteq S' \times Msg \times Msg \times S'$ is the least relation such that $(s_1, s_2) \xrightarrow{\mu_a, \mu_b}{}' (s'_1, s'_2)$ if $s_1 \xrightarrow{\mu_a, \mu_b}_1 s'_1$ and $s_2 \xrightarrow{\mu_a, \mu_b}_2 s'_2$.*

As the only requirement to have a well-formed state-based model is the call stack property, it is enough to build a restrictor model which induces traces following the call stack property.

Definition 7 (Call Stack Restrictor Model). *Let $Header \subseteq Obj \times Mtd$ be the set of all message headers. The* call stack restrictor model *$\mathcal{C} = \langle S_c, \Theta_c, s_{0,c} \rangle$ is a state-based model whose state is a stack of elements of Header, with the initial state being the empty stack[5]. The transition $s_1 \xrightarrow{\mu_a, \mu_b}_c s_2$ exists if one of the following conditions hold.*

[5] We denote the empty stack as ϵ and a non-empty stack as $h' = h : hs$, where h is the top element and hs denotes the rest of the stack.

1. $dir(\mu_a) = \rightarrow \wedge\ dir(\mu_b) = \rightarrow \wedge\ s_1 = hs \wedge s_2 = header(\mu_b) : header(\mu_a) : hs$
2. $dir(\mu_a) = \leftarrow \wedge\ dir(\mu_b) = \leftarrow \wedge\ s_1 = header(\mu_a) : header(\mu_b) : hs \wedge s_2 = hs$
3. $dir(\mu_a) = \rightarrow \wedge\ dir(\mu_b) = \leftarrow \wedge\ header(\mu_a) = header(\mu_b) \wedge s_1 = s_2$
4. $dir(\mu_a) = \leftarrow \wedge\ dir(\mu_b) = \rightarrow \wedge\ s_1 = header(\mu_a) : hs \wedge s_2 = header(\mu_b) : hs$

Note that our restrictor model is non-deterministic. However, since our state-based model is deterministic, the product will remain deterministic.

Definition 8 (Well-formed State-based Model). *Given a state-based model $\mathcal{M}$, the* well-formed state-based *model for $\mathcal{M}$ is $\mathcal{M}_{wf} = \mathcal{M} \otimes \mathcal{C}$.*

Example (Well-formed state-based model of the `Subject` component). The well-formed state-based model of the example is $\mathcal{M}_{wfsubj} = \mathcal{M}_{wfsubj} \otimes \mathcal{C}_{subj}$. In order to ensure the call stack property, the states of $\mathcal{M}_{wfsubj}$ are thus a subset of `Subject` × `Observer` × `Observer` × `Stack<State>` × `Stack<Header>`.

5 Model Abstraction

We are interested in reducing the size of our models without altering the component trace sets represented by the models. To do this, the models need to be related with each other. One such relation is the state abstraction relation (cf. §7.4 of [12]), which is an instance of the simulation relation [13]. We extend this relation to our setting and reuse known results to build a most abstract model.

Definition 9 (State Abstraction). *Given two state-based component models $\mathcal{M}_i = \langle S_i, \Theta_i, s_{0,i} \rangle$, $i = 1, 2$, we say that $\mathcal{M}_2$ is* more abstract *than $\mathcal{M}_1$ iff there is a total abstraction function $\alpha : S_1 \rightarrow S_2$ such that*
$s_{0,2} = \alpha(s_{0,1})$*; and if* $s_1 \xrightarrow{\mu_a, \mu_b}_1 s_1'$*, then* $\alpha(s_1) \xrightarrow{\mu_a, \mu_b}_2 \alpha(s_1')$.

Example (Abstraction of the trace-based model of the `Subject` component). Using Lemma 1, we build $\mathcal{M}_{tsubj}$ as the state-based model from the trace-based model $\mathcal{T}_{subj}$. We compare $\mathcal{M}_{tsubj}$ and $\mathcal{M}_{wfsubj}$ by providing an abstraction function $\alpha : S_{tsubj} \rightarrow S_{wfsubj}$ as follows.

$\alpha(\epsilon) =$ (**null**, **null**, **null**, **null**, ϵ)

$\alpha((\rightarrow sbj.\texttt{Subject}(o_1, o_2), \leftarrow sbj.\texttt{Subject}()) : \epsilon) =$
(*sbj*, o_1, o_2, ϵ, ϵ)

$\alpha((\rightarrow sbj.\texttt{update}(s), \rightarrow o.\texttt{notify}(s)) : l) =$
(*sbj*, o_1, o_2, s : sl, o.`notify`:*sbj*.`update`:hl)
where (*sbj*, o_1, o_2, sl, hl) $= \alpha(l)$, $o = o_1$

$\alpha((\leftarrow o.\texttt{notify}(), \rightarrow p.\texttt{notify}(st)) : l) =$ (*sbj*, o_1, o_2, s : sl, p.`notify`:hl)
where (*sbj*, o_1, o_2, s : sl, o.`notify`:hl) $= \alpha(l)$, $o = o_1$, $p = o_2$, $st = s$

$\alpha((\leftarrow o.\texttt{notify}(), \leftarrow sbj.\texttt{update}()) : l) =$ (*sbj*, o_1, o_2, sl, hl)
where (*sbj*, o_1, o_2, s : sl, o.`notify`:*sbj*.`update`:hl) $= \alpha(l)$, $o = o_2$

For the initial state, we map the empty history to **null**s and the empty header stack. Each of the remaining cases shows a one-to-one correspondence between the postconditions defined in the trace-based specification (Fig. 3) and the postconditions defined in the state-based specification (Fig. 2).

Because such an abstraction function α exists, we can conclude that the trace-based model of the `Subject` component is more abstract than the state-based model of the `Subject` component derived from the state-based specification. The following main theorem states this result in a more general way.

Theorem 1. *For any trace-based model $\mathcal{T}$, there is a well-formed state-based model $\mathcal{M}_{wf}$ which is more abstract than $\mathcal{T}$. The converse does not hold.*

Proof (sketch). The first part of this lemma follows directly from Lemma 1 and taking the identity function as α. For the converse, consider a specification which concentrates only on the well-formedness property of the traces. Any well-formed trace in the trace-based model of the specfication which has completed all updates can be abstracted to the state with empty stacks in the state-based model. As the abstraction must be a function, the converse does not hold.

We can go further than the main theorem by adapting well-known results about state transition systems. We first define the standard notion of simulation.

Definition 10 (Simulation [13]). *Let $\mathcal{M}_i = \langle S_i, \Theta_i, s_{0,i} \rangle$, $i = 1, 2$ be state-based models over Msg. A* simulation *for $(\mathcal{M}_1, \mathcal{M}_2)$ is a binary relation $\mathcal{R} \subseteq S_1 \times S_2$ such that $(s_{0,1}, s_{0,2}) \in \mathcal{R}$; and if $(s_1, s_2) \in \mathcal{R}$ and $s_1 \xrightarrow{\mu_a, \mu_b}_1 s_1'$, then there is $s_2' \in S_2$ such that $s_2 \xrightarrow{\mu_a, \mu_b}_2 s_2'$ and $(s_1', s_2') \in \mathcal{R}$.*

If there exists an simulation relation $\mathcal{R}$ for $(\mathcal{M}_1, \mathcal{M}_2)$, we say that $\mathcal{M}_1$ is *simulated* by $\mathcal{M}_2$, denoted $\mathcal{M}_1 \preceq \mathcal{M}_2$. Furthermore, if the simulation relation occurs in both directions (i.e., $\mathcal{M}_1 \preceq \mathcal{M}_2$ and $\mathcal{M}_2 \preceq \mathcal{M}_1$), we say that $\mathcal{M}_1$ is *simulation equivalent* to $\mathcal{M}_2$, denoted by $\mathcal{M}_1 \simeq \mathcal{M}_2$. We are only interested in the (unique [14]) *maximal* simulation relation, which is a simulation relation that contains all other simulation relations between the two models.

The following lemma shows that the abstraction is a simulation.

Lemma 2. *If $\mathcal{M}_2$ is more abstract than $\mathcal{M}_1$, then $\mathcal{M}_1 \preceq \mathcal{M}_2$.*

Proof. Consider $\mathcal{M}_i = \langle S_i, \Theta_i, s_{0,i} \rangle$, $i = 1, 2$ and $\alpha : S_1 \to S_2$ to be the abstraction function. We simply take $\mathcal{R} = \{(s, \alpha(s)) \mid s \in S_1\}$.

A most abstract model is a model which up to renaming has no more abstraction. To construct it, we need the notion of equivalence classes.

Definition 11 (Equivalence Classes). *Let S be a set and $\mathcal{R}$ an equivalence on S. For $s \in S, [s]_{\mathcal{R}} = \{s' \mid (s, s') \in \mathcal{R}\}$. The quotient space of S under $\mathcal{R}$ is defined as $S/\mathcal{R} = \{[s]_{\mathcal{R}} \mid s \in S\}$.*

By building the equivalence classes of the states based on the simulation relation, we end up with the simulation quotient model.

Definition 12 (Simulation Quotient Model). *Given a state-based model $\mathcal{M} = \langle S, \Theta, s_0 \rangle$, the* simulation quotient state-based model *$\mathcal{M}/\simeq$ is a triple $\langle S/\simeq, \Theta_{\simeq}, [s]_{\simeq} \rangle$, where $[s] \xrightarrow{\mu_a, \mu_b}_{\simeq} [s']$ if $s \xrightarrow{\mu_a, \mu_b} s'$.*

Grumberg and Bustan [15] show that the simulation quotient model $\mathcal{M}/\simeq$ is the unique smallest (in terms of number of states) state-based model up to renaming which is simulation equivalent to $\mathcal{M}$. Since the quotient space is a partitioning of the original state set S, we can also see it as an abstraction.

Theorem 2 (Most Abstract Model [15]). *The simulation quotient model $\mathcal{M}/\simeq$ is the most abstract model of the state-based model $\mathcal{M}$ up to renaming.*

Example (Simulation quotient model of the `Subject` *component).* The following lemma states that the well-formed state-based model of the `Subject` component specification is a simulation quotient model for the specification.

Lemma 3. *$\mathcal{M}_{wfsubj}$ is the simulation quotient model for $\mathcal{M}_{tsubj}$.*

Proof (sketch). By contradiction. Assume $\mathcal{M}_{wfsubj}$ is not the simulation quotient model. Then there exist two states of $\mathcal{M}_{wfsubj}$ that are equivalent, which is false.

By Theorem 2, $\mathcal{M}_{wfsubj}$ is the unique smallest state-based model for the `Subject` component specifications and, hence, its most abstract model.

In the deterministic setting, the notions of simulation equivalence and trace equivalence collapse [16]. As a result, the quotient model is a most abstract model, i.e., a model such that there is no other smaller state-based model that is more abstract and still retains exactly the same behavior.

In a non-deterministic setting, the abstraction function defined in Def. 9 only guarantees finite trace inclusion, while the simulation equivalence guarantees finite trace equivalence [17]. In general if we want a more abstract model to remain behaviorally equivalent to the abstracted model, then we need to use bisimulation equivalence [18] as the reduction. This requires a more general definition of abstraction, where, in addition to the definition of abstraction given here, the abstract states need to be related back to the abstracted states.

6 Related Work

State-based models are usually, in the object-oriented setting, specified using contracts as introduced in Eiffel [3]. JML [4] and Spec# [19] adopt this approach to allow modular specifications in Java and C#, respectively. Contracts mainly consist of method pre-/postconditions and class invariants, making it non-trivial to deal directly with callbacks as in the subject/observer example (Sect. 2).

A generalized state-based model similar to the one given here is present in the Z^{++} methodology [20], where object-oriented real-time systems are specified using real-time logic. While similar notions of message predicates are used within the state-based specifications, their purpose is for timing measurement.

Trace-based specifications are well-known in the literature of processes and modules [21,22]. This trace idea has also appeared in the object-oriented setting as early as [8] in terms of communication histories, further developed in [23] and later on in [24,25], especially for facilitating the modeling of open asynchronous distributed systems.

In connection to proof systems, trace-based specifications have been used most recently in [9,26] in the Creol [27] setting. The trace of a component is seen as a projection from the global trace and checked against invariants which capture communication patterns in form of regular expressions of messages.

At the program level, the idea to characterize the behavior of object-oriented components by admissible traces is also used in the context of observational equivalence to give a fully abstract semantics for object-oriented languages [5,28].

Jass [29] gives specifications in the form of trace assertions, whose semantics is based on CSP. This technique allows to specify our subject example in a similar fashion. However, a clear component model is missing. Similarly, Cheon and Perumandla [30] extend JML to introduce assertions based on call sequences in form of regular expressions, by abstracting from argument values and callee.

7 Conclusion and Future Work

In this paper we have shown for a sequential object-based setting that state-based models are abstractions of trace-based models. We have also given an example of a subject component illustrating the relation between state- and trace-based specification approaches by describing them in a common framework.

As the subtitle of this paper suggests, our long-term goal is to formalize this generalization to describe precisely the behavior of a component in an object-oriented setting. We are especially interested in defining the connection between such mixed trace- and state-based specifications and programming languages. An ideal connection would allow specification composition which in turn allows modular verification.

Another natural extension to this work is to consider a more general setting such as full object-orientation (i.e., allowing subtyping and inheritance) and concurrency. In addition, sometimes we would like to specify how the component should be used. For example, we want to guarantee that the observer, upon being notified, actually responds to the subject component (i.e., does not diverge). By specifying such restrictions about the environment, the set of traces is no longer prefix-closed, and thus the semantics of our state-based model has to be altered.

Acknowledgements. The authors would like to thank the anonymous referees for their constructive comments and suggestions.

References

1. Müller, P.: Modular Specification and Verification of Object-Oriented Programs. PhD thesis, FernUniversität Hagen (2001)
2. Cheon, Y., Leavens, G., Sitaraman, M., Edwards, S.: Model variables: Cleanly supporting abstraction in design by contract. Softw. Pract. Exper. **35**(6) (2005) 583–599
3. Meyer, B.: Object-Oriented Software Construction. Prentice-Hall (1988)
4. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: A behavioral interface specification language for Java. ACM SIGSOFT Software Engineering Notes **31**(3) (2006) 1–38

5. Jeffrey, A., Rathke, J.: Java Jr: Fully abstract trace semantics for a core Java languages. In: ESOP. Volume 3444 of LNCS., Springer (2005) 423–438
6. Kyas, M.: Verifying OCL Specifications of UML Models: Tool Support and Compositionality. PhD thesis, Leiden University (2006)
7. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Addison-Wesley, Boston, MA (January 1995)
8. Dahl, O.J.: Can program proving be made practical? In: Les Fondements de la Programmation. (1977) 57–114
9. Dovland, J., Johnsen, E.B., Owe, O.: Observable behavior of dynamic systems: Component reasoning for concurrent objects. ENTCS **203**(3) (2008) 19–34
10. Kyas, M., de Boer, F.S., de Roever, W.P.: A compositional trace logic for behavioural interface specifications. NJC **12**(2) (2005) 116–132
11. Fredkin, E.: Trie memory. CACM **3**(9) (1960) 490–499
12. Baier, C., Katoen, J.P.: Principles of Model Checking (Representation and Mind Series). The MIT Press (2008)
13. Milner, R.: An algebraic definition of simulation between programs. In: IJCAI. (1971) 481–489
14. Grumberg, O., Long, D.E.: Model checking and modular verification. ACM TOPLAS **16**(3) (1994) 843–871
15. Bustan, D., Grumberg, O.: Simulation-based minimazation. ACM TOCL **4**(2) (2003) 181–206
16. Kucera, A., Mayr, R.: Simulation preorder over simple process algebras. Information and Computation **173**(2) (2002) 184–198
17. van Glabbeek, R.J.: The linear time-branching time spectrum (extended abstract). In: CONCUR. Volume 458 of LNCS., Springer (1990) 278–297
18. Park, D.: Concurrency and automata on infinite sequences. In: Proceedings of the 5th GI-Conference on TCS, London, UK, Springer-Verlag (1981) 167–183
19. Barnett, M., Leino, K.R.M., Rustan, K., Leino, M., Schulte, W.: The Spec# programming system: An overview. In: CASSIS, Springer (2004) 49–69
20. Lano, K.: Formal Object-Oriented Development. Springer-Verlag New York, Inc., Secaucus, NJ, USA (1995)
21. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall (1985)
22. Bartussek, W., Parnas, D.L.: Using assertions about traces to write abstract specifications for software modules. In: ECI. Volume 65 of LNCS., Springer (1978) 211–236
23. Nierstrasz, O.: Regular types for active objects. In: OOPSLA. (1993) 1–15
24. Johnsen, E.B., Owe, O.: A compositional formalism for object viewpoints. In: FMOODS. Volume 209 of IFIP Conference Proceedings., Kluwer (2002) 45–60
25. Kyas, M., de Boer, F.S.: On message specification in OCL. In: Compositional Verification in UML. Volume 101 of ENTCS., Elsevier (2004) 73–93
26. Ahrendt, W., Dylla, M.: A verification system for distributed objects with asynchronous method calls. In: ICFEM. Volume 5885 of LNCS., Springer (2009) 387–406
27. Johnsen, E.B., Owe, O.: An asynchronous communication model for distributed concurrent objects. Software and System Modeling **6**(1) (2007) 39–58
28. Steffen, M.: Object-connectivity and observability for class-based, object-oriented languages. (2006) Habilitation Thesis, University of Kiel.
29. Bartetzko, D., Fischer, C., Möller, M., Wehrheim, H.: Jass - Java with assertions. ENTCS **55**(2) (2001) 1–15
30. Cheon, Y., Perumandla, A.: Specifying and checking method call sequences of Java programs. Software Quality Journal **15**(1) (2007) 7–25

Controlling the Unknown

Casandra Holotescu

Politehnica University of Timişoara
Dept. of Computer and Software Engineering
casandra@cs.upt.ro

Abstract. A lot of work has been done in the area of building component-based systems with correct-by-construction adaptors. This is accomplished by using preexisting specifications of the component behaviour. But what happens when known components get to interact with incompletely specified, black-box components? How can we avoid errors in this kind of system, without modifying existing/legacy components? We present a method to explore and control such systems. Our approach exploits information in both correct and erroneous runs to build a controller that ensures our system will avoid observed errors when interacting with the unspecified component. We consider the behavioural specifications for our known, legacy component to be previously documented and we infer partial behaviour information of the unknown component by studying its reactions to various interaction scenarios.

Keywords: component-based systems, model refinement, adaptation, maintenance

1 Introduction

Component-based software engineering aims to improve software productivity by assembling large systems out of reusable components. However, as these components are often developed separately, the risk of mismatching appears at different levels: signature, behaviour, quality of services, etc. This inconvenience is addressed by design-time adaptation.

In the area of behavioural component adaptation, an adaptor is considered to be a specific component-in-the-middle that would properly coordinate the interactions between components towards the desired functionality. Several approaches for automatically generating correct by construction adaptors for system composition have been proposed, e.g. by Schmidt and Reussner [20], Autili et al. [3] or Canal et al. [7]. While earlier pioneering approaches, such as [21] or [20] though already semiautomatic, have only focused on ensuring a non-deadlocking interaction among components, later research such as [3], [7] has resulted in tools able to automatically synthesize adaptors that mediate the interactions in the component based system, so that its resulting behaviour satisfies a temporal logic property (described as a Büchi automaton, vector transition system, finite state machine, etc.). If both the system behaviour and the goal property are

regarded as automata, the resulting system automaton must be a simulation of the goal.

There is a close relation between component adaptation and control theory [19]. Considering the desired functionality and absence of deadlock as part of the system specification, we may view an adaptor as a controller over the system plant, using message forwarding or consumption as a means to enable and disable behaviour in the plant.

The above approaches assume all component behaviours are known, either prespecified as finite state machines, labeled transition systems or Petri nets, or derived in this form from more intuitive, visual formalisms such as message sequence charts. However, this is not always the case in the real component-based software development environment, and it is not uncommon for a system architect to have to integrate a black-box component. Integrating a component with insufficient behavioural specification is difficult, as none of the existing automatic solutions for adaptor synthesis can be applied, and the resulting system can be quite vulnerable to errors. Resolving an observed error in this kind of system, by automatically generating a correctness-enforcing adaptor, is the main aim of our method.

For simplification, consider a component with available behavioural specification, which interacts with another, black-box component, whose behaviour is partially unspecified. The interaction takes place by asynchronous message exchange. The resulting system must satisfy a certain property and also avoid deadlock. At runtime, a certain set of interactions results in an error, i.e., property violation, deadlock or system crash. As the latter two can be subsumed by the former one, we'll only consider temporal property violations from now on. Let us also assume we have a failing run trace.

Our method consists of the following steps (fig.1):

- Behaviour exploration and refinement: the next steps are repeated until a satisfying refinement is found or a cost limit is reached.
  - Exploration: We introduce a "fake" adaptor among the two components with the purpose of conducting experiments with different execution scenarios. Its actions are directed towards exploring the system state space instead of performing a real, correctness-ensuring adaptation.
  - Refinement: A tentative model of the unspecified component is constructed and incrementally refined, using the information obtained during the previous exploration phase. The component is assumed to have a deterministic behaviour.
- Synthesis: using the refined approximation of the unknown component, we synthesize an adaptor that will forbid the erroneous interaction sequences.

The exploration process takes repeatedly the following steps, as long as there are unexplored controllable choices:

- We launch again the system execution and follow the error trace up to its last controllable and incompletely explored decision point.
- We repeat following actions until execution completes:

- The fake adaptor initiates a bounded model-checking phase over the system model, that will search for the best next steps towards property satisfaction.
- The best choice yet unexplored is enabled by the fake adaptor in the concrete system execution.
- Unknown component reaction is observed.

– Execution ends by success/failure and its trace is stored.

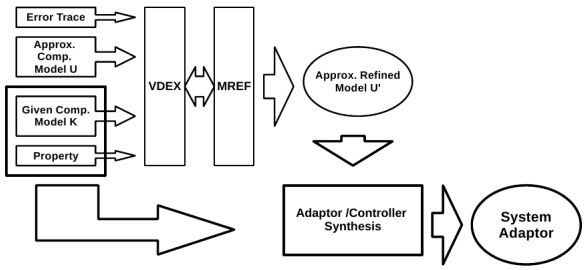


Fig. 1: Main method steps: verification-driven execution, model refinement and controller synthesis

The verification driven execution follows each time the initial error trace, exploring different options on the path by enabling and disabling controllable events. This means that also the model refinement, as it employs the use of information obtained during the runtime exploration, is directed towards clarifying the behavioural vicinity of the error trace.

Once the approximated model is refined enough so that all control points on the error trace have been completely explored through verification driven execution, the refinement process ends. We have now experimented with all potential controllable events on the path to error, and we can decide on the correctness ensuring controller synthesis.

2 Method

2.1 Assumptions

We have a real component-based system S_R, composed from C_K, the known component, and C_U, the unknown one.

Let us assume component C_K has a previously specified behaviour, described by a finite state machine: a tuple $K = \langle Q^K, Q_f^K, q_0^K, \Sigma^K, \delta^K \rangle$, where Q^K is the set of states, $Q_f^K \subseteq Q^K$ the set of final states, q_0^K is the initial state, Σ^K a set of events of the message send/receive type: $msg!$ and $msg?$ and $\delta^K : Q^K \times \Sigma^K \rightarrow Q^K$ the partial transition function.

We also associate the unknown component C_U with a tentative finite state machine $U = \langle Q^U, Q_f^U, q_0^U, \Sigma^U, \delta^U \rangle$. Its event set Σ^U is assumed to be the mirroring of Σ^K, thus $\Sigma^U = \{msg!|msg? \in \Sigma^K\} \cup \{msg?|msg! \in \Sigma^K\}$. As we have no information on this finite state machine, we start by considering it as most general and its transition function becomes $\delta^U : Q^U \times \Sigma^U \rightarrow \mathcal{P}(Q^U)$, thus allowing for nondeterminism. We will assume, for now, that $Q^U = Q_f^U = \{q_0^U\}$ and $\delta^U\left(q_0^U, \sigma\right) = \{q_0^U\}$ is defined for all $\sigma \in \Sigma^U$. It is important to remember that U is not modelling the real behaviour of component C_U, but an overapproximation of it - although C_U is deterministic, by allowing U to be nondeterministic we can make sure that during the refinement process U will always overapproximate the behaviour of C_U.

We consider the interactions between components as asynchronous, by means of bounded buffers. Let $K \times U$ be their asynchronous composition. Each component has an output and an input buffer, both of length l. The input buffer of one component is directly connected to the output buffer of the other, as no adaptor exists between the two components - the behaviour of the second component is unknown, so we cannot synthesize an adaptor by any of the classic methods. When external control is performed on the system, this actually happens by controlling the message transfer between the input and output buffers.

All events in the event sets Σ_K and Σ_U, i.e., reading and writing to/from buffers, are assumed observable. Thus, if component C_U receives a message msg, the receive event $\sigma = msg?$ can be externally observed.

The ideal system S to be obtained from the composition of K and U complies with the property φ. The desired property φ is also expressed by means of a finite state machine $\langle Q^\varphi, Q_f^\varphi, q_0^\varphi, \Sigma^\varphi, \delta^\varphi \rangle$, where $\Sigma^\varphi \subseteq \Sigma^K \cup \Sigma^U$. For simplification, the property φ also includes the non-deadlocking requirement, thus $\varphi = \phi \times \rho$, where ϕ is the functional specification of the system and ρ the specification of deadlock avoidance. We consider compliance as simulation: $S \preceq \varphi$.

The real system S_R is found to violate property φ at runtime. We assume we have one recorded trace of an erroneous execution, as a sequence of events $t = \sigma_1\sigma_2...\sigma_n$, with $\sigma_i \in \Sigma^K \cup \Sigma^U$.

We aim to build a controller that restricts the behaviour of system S_R such that the error observed in t no longer occurs. In a component-based system, such a controller is an adaptor that enables events by message forwarding, and disables them by message consumption, thus controlling the message transfer between component buffers. Therefore, while the receive events $\sigma \in \Sigma_?$ are controllable by the adaptor, send events $\sigma \in \Sigma_!$ are uncontrollable.

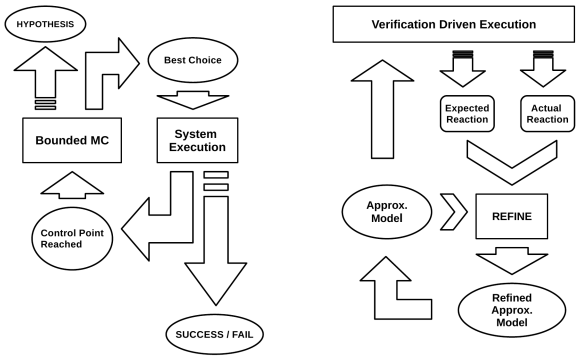
2.2 Discovering behaviour

We start by assuming the behaviour of the unknown component as most general. We cannot build a controller for system S_R starting from this overapproximated model of C_U, since it would imply using a large number of virtual, possibly spurious, control sequences. How can we refine this abstraction in order to make it more precise and thus more useful? This subsection will deal with the issue of

discovering new behaviour of component C_U, by exploring its real behaviour at runtime and correspondingly refining the tentative finite state machine U (fig.2).

We use verification-driven execution: we verify the desired property φ on the modelled system $K \times U$ by means of bounded model-checking [5], while directing the execution of the real system S_R based on the information provided by model checking. This enables us to verify the model U of component C_U through actual trace execution and, when inconsistencies appear, to appropriately refine it. To perform these operations a "fake" adaptor - an interactive mediating component, will be inserted between the two components.

As our approach aims to find a controller that enables the system to avoid the observed error, the refinement process is oriented on obtaining a reapproximation of U useful in the controller synthesis. Thus, the exploration is also centered on the error trace t: verification driven execution exploits t, and tries to find successful controllable executions in S_R, from various prefixes of t. The exploration and refinement process stops when all controllable decision points of t have been completely explored by verification driven execution.



(a) VDEX (b) MREF

Fig. 2: Behaviour exploration and refinement (a) Verification driven execution (b) Model refinement

Verification-driven execution Let $\overline{L(S)}$ be the prefix-closed language of the ideal system $S = (K \times U) \parallel \varphi$ that complies to the property φ. Let $pre(t)$ be the longest prefix of t that complies to φ, $pre(t) = t_k$ such that $t_k = \sigma_1\sigma_2 \ldots \sigma_k \in \overline{L(S)}$, but $t_{k+1} \notin \overline{L(S)}$. In order to disable t, we have to prevent the event σ from occurring by disabling it, if controllable, or by disabling its previous controllable event in the error trace.

We define a control point in $K \times U$ as a state q for which at least one of the outgoing controllable event sets $\Sigma^K_?(q)$ and $\Sigma^U_?(q)$ contains more than one

controllable event: $|\Sigma_?^K(q)| \geq 2 \vee |\Sigma_?^U(q)| \geq 2$. For a controllable event to be disabled, an alternative controllable, receive event $msg?$ has to be enabled in the control point. This can only happen if the expected message msg is already available.

Let now q_{cp} be the last control point in $K \times U$ that precedes the error-inducing event σ in the error trace t.

We want to discover possible new behaviours of the real system S_R starting from the control point q_{cp}. We will explore the hypothetical alternative choices from q_{cp} in the system model $K \times U$ by means of bounded model checking, with b the maximum depth bound. The finite state machine describing the property φ will be executed synchronously with the system model $K \times U$.

We launch the run, and force the system to perform the same sequence of interactions executed in the original error trace, until the execution reaches state q_{cp}. Due to the uncontrollability of send events, this might not always be possible. If this happens and the new trace t' is no longer a prefix of t, $t' \notin Pre(t)$, then the next control point on t' becomes the new q_{cp} for that execution. From the control point q_{cp} of the system $K \times U$ and the corresponding state q_{cp}^{φ} of the system specification, we start a depth-first exploration of the alternative choices offered by q_{cp}.

Let us now focus on the possible traces of maximum length b, generated by enabling an alternative choice σ at the control point q_{cp}. Let $|pre(t_i)|$ be the length of a possible trace t_i, which is enabled from q_{cp} through the controllable event σ_i, such that $pre(t_i)$ satisfies φ. The trace t_i so that $|pre(t_i)| \geq |pre(t_j)|$, $\forall j \neq i$ is considered the best unexplored trace from q_{cp}, thus its corresponding choice σ_i is the best next step from q_{cp}. Therefore, the fake adaptor enables σ_i for the on-going execution.

In this way, a decision based on a φ verification lookahead step is to be taken at each control point met on its way by the execution, thus a controllable event σ being enabled. The mediated execution of the system continues until either the violation of the desired property φ cannot be avoided, or the execution blocks, or it successfully completes.

After one control point q_{cp} has been completely explored, its previous control point on t will follow. The verification-driven execution phase ends when the U model cannot be further refined, since all control points on t in the system model have been explored. The performed system executions are partitioned into two sets: T_c, the set of correct execution traces, and T_e, the set of erroneous traces.

Approximation refinement As the finite state machine U describing the behaviour of C_U is overapproximated, the following situation will appear, especially in the first runs: the fake adaptor sends message msg to C_U, but C_U doesn't accept it. Thus, we have discovered a difference between U and the real behaviour of C_U, therefore we need to refine U.

One individual refinement step takes place as follows. The set $\Sigma^U(q)$ of events is enabled for the current state q.

Case 1. Let us suppose that after an event σ_i in state q_1 of U, the following controllable event σ_{i+1}, assumed by the fake adaptor to happen, fails to do so. That means the component C_U has got into a state in which σ_{i+1} is not enabled anymore. This state, previously thought to be q, must now be replaced with a new copy q'. The transition function becomes: $\delta^U(q_1, \sigma_i) = \{q'\}$, $\delta^U(q', \sigma) = \delta^U(q, \sigma)$, $\forall \sigma \in \Sigma^U \setminus \{\sigma_{i+1}\}$. A self-loop on σ from q will lead in q' to the transitions $\delta^U(q', \sigma) = \{q, q'\}$.

Case 2. If from a state q we have nondeterminism in the model U: $|\delta^U(q, \sigma_i)| \geq 2$ and we observe an execution sequence $\sigma_i.\sigma_{i+1}$ from q, we can use this execution to resolve the nondeterminism, as component C_U is assumed deterministic. Thus, all transitions $\delta^U(q, \sigma_i)$ to a state q' for which $\delta^U(q', \sigma_{i+1}) = \emptyset$ are removed from U.

Thus, by always splitting the current state and/or reducing the number of outgoing transitions from a state, the C_U component model U is incrementally refined during the exploratory training phase. The refinement stops either when state number limit λ has been reached, or all the control points on t have been explored. We obtain a model U' that is a more precise approximation of the real component C_U.

The model U' might still contain, however, nondeterministic transitions triggered by uncontrollable events σ, from states q where σ was neither observed at execution, nor eliminated from $\Sigma^U(q)$. We propose three ways to resolve this situation, depending on the system security requirements.

- **Optimistic approach**: For all states q, all transitions triggered by uncontrollable events $\sigma \in \Sigma_!^{U'}(q)$ not observed at execution are removed from U. Thus, the transition function becomes $\delta^{U''}(q, \sigma) = \emptyset$ iff $\sigma \in \Sigma_!^{U'}(q)$ and $observed(q, \sigma) = false$. We obtain a deterministic refined model U'', that leads to a small controller, but further runtime risks may appear if σ can actually manifest from q.
- **Pessimistic approach**: The final refined model $U'' := U'$, and the nondeterminism due to uncontrollable events unobserved at execution remains. This leads to a larger, more difficult to synthesize controller, but the resulting system will be safe.
- **Semi-optimistic approach**: For any state q, all uncontrollable events $\sigma \in \Sigma_!^{U'}(q)$ not observed during the verification-driven execution phase, are assumed unobservable. The transition function becomes $\delta^{U''}(q, \sigma) = \{q\}$ iff $\sigma \in \Sigma_!^{U'}(q)$ and $observed(q, \sigma) = false$. We obtain a deterministic refined model U'', that still accepts all $\sigma \in \Sigma_!^{U'}(q)$. The resulting controller, while significantly smaller than the pessimistic one, is still much larger than the optimistic controller. Further risks may still appear if for some such σ the actual transition from q is not a self loop, however the system is safer than in the optimistic case.

Our model refinement phase is related to the counterexample guided abstraction refinement (CEGAR) developed by Clarke et al. [8], since the blocking execution can be regarded as a spurious counterexample. The difference is that

we do not have an accessible concrete model for the refinement of the overapproximated model, but a black-box component, which we explore at runtime.

2.3 Controller Synthesis

An adaptor, as earlier stated, acts as a controller in a component-based system: it can enable receive events by forwarding the corresponding messages, or disable them by consuming these messages. Considering the set of send events $\Sigma_!$ in a component finite state machine as uncontrollable, and the set of receive events $\Sigma_?$ as controllable, we can solve the control problem in our system by building an adaptor A. The plant to be controlled will be the asynchronous product $K \times U''$ and the specification will be property φ.

In order to ensure that the real system composed from the components C_K and C_U will satisfy φ, as the model U'' still represents an approximation of C_U, two possible variants of controller synthesis can be employed. The first, classical one, allows the construction of a most permissive controller, while the second one, more restrictive, limits the system to the observed correct and controllable behaviour.

Permissive control Let us note by $Ctrl$ the system controller. For the computation of $Ctrl$ as the controller of the plant $K \times U''$, for the specification φ we will use the classical result of Ramadge and Wonham: $Ctrl = \mathbf{supcon}(K \times U'', \varphi)$, where **supcon**, described in [19], is a fixpoint procedure.

This leads us to a most permissive controller $Ctrl$, where all behaviours of the plant $K \times U''$ that does not end up in a violation of property φ is allowed. The same result, as shown in [4, 2] can be obtained if the control problem is modelled as an acceptance game, as the most permissive winning strategy by an alternating reachability algorithm. If the specification φ is translated to an ATL formula ψ, within a system with two agents, one determined by the set of controllable transitions, and the other one by the set of uncontrollable ones, the desired controller is obtained through model checking of the formula $\ll Ctrl \gg \mathbf{G}\psi$. [1]

The adaptor A is then obtained directly from the controller $Ctrl$ by mirroring in the set Σ^A all events from Σ^{Ctrl}.

Restrictive control If, out of cost reasons, the behaviour exploration process has been stopped early, the resulting model U'' might still exhibit some false controllability, i.e., transitions $\delta^{U''}(q, \sigma)$, where $\sigma \in \Sigma_?^{U''}$, that do not manifest at runtime. This may lead to execution errors in the controlled system. For many systems, this can be accepted: the run is stopped, the model U'' re-refined to a new model U''', and another adaptor A' is synthesized. However, for systems with strong security constraints, a more restrictive solution is needed, with the price of sacrificing part of the allowed behaviour.

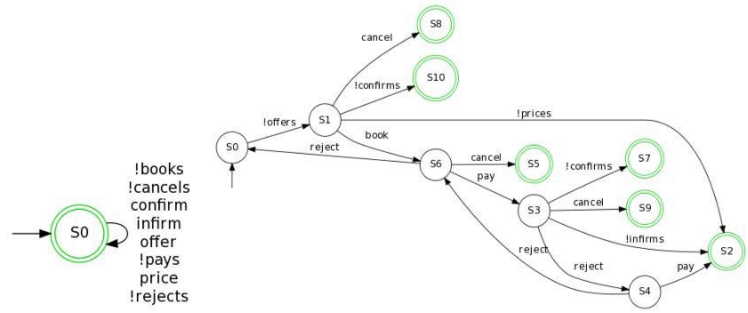
During the exploration of system S_R behaviour, two sets of execution traces have been obtained: the set T_c containing all correct traces, and T_e containing all erroneous execution traces. Let $T_{cctrl} \subseteq T_c$ be the set of all correct and

controllable execution traces. If the behaviour exploration of S_R has been wide enough, there are sufficient correct controllable traces in T_{cctrl} for us to limit the controlled system behaviour to them without being excessively restricting.

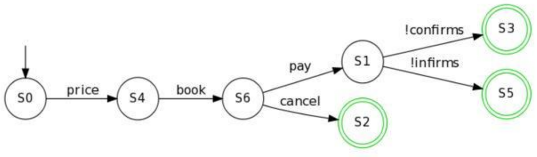
Let C be the finite state machine resulting from merging all the traces in T_{cctrl}. The desired restrictive controller for the system is then $CSafe = C$, while the adaptor A is obtained from *CSafe* by mirroring the event set.

3 Model refinement and controller synthesis example

Suppose we have a system composed from the two components in figure 3. The component C_K is a ticket-selling server, while the component C_U is a client terminal. The behaviour of C_U is unspecified, therefore it is approximated by the most general finite state machine U. The system specification, represented in figure 4 is ϕ, the functional specification. For simplification, the non-deadlock property is omitted. Also, suppose the following trace violates the functional specification ϕ: $t = offer?.book!.price?.pays!.offers!.book?$, where ϕ is satisfied by the prefix $offer?.book!.price?.pays!.offers!.$



(a) $U\ FSM$ (b) $K\ FSM$



(c) $\phi\ FSM$

Fig. 3: Component and specification models (a) Unknown component initial tentative model and (b) Known component model (c) System specification

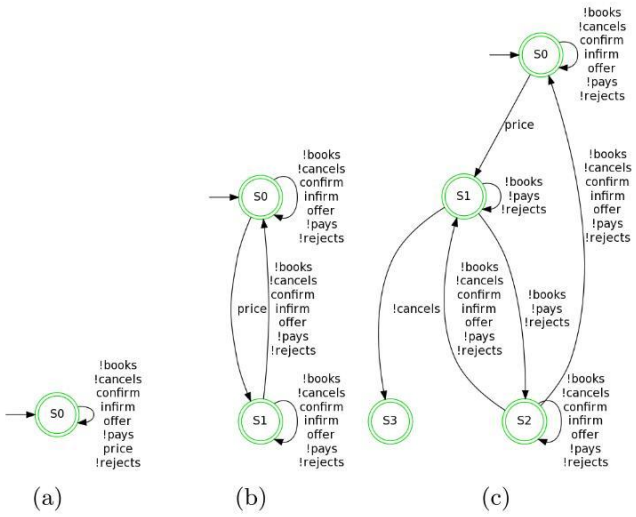


Fig. 4: Refinement process

In figures 4 and 5 we can observe the refinement process:

- 4(a): execution reaches prefix $offer!.offers?.book!.book?.price!.price?$, message *price* is sent again to C_U and not accepted. Thus, a transition has taken place at the previous event, and the component is in a new state. The model is refined to 4(b). The original state was accepting, so is the one resulted from the split.
- 4(c): message *cancel* is received from C_U by the fake adaptor. After *cancel*! component C_U has reached a final state and the split is performed.
- 5(a): after same prefix, message *pay* is received, then a control point is reached where choices $confirm?$ and $infirm?$ are explored in two different executions. Messages $confirm$ and $infirm$ are both accepted and they both lead to final states. In a further execution, from same state message *cancel* is received and a final state is reached.
- 5(b): We try to enable choices $confirm$, $infirm$ from the initial state, but fail. We have explored all control points on t, all controllable event hypotheses have been tested.
- 5(c): Only accepting states found to be final model states are kept marked.

The resulting refined model in 5(c) still has nondeterministic transitions, due to unobserved potential events, that are all uncontrollable. There are three ways to deal with this situation in order to build a controller. Figure 6 presents the optimistic approach. The resulting controller is rather small. The semi-optimistic approach is presented in figure 7 together with its controller. Finally, the pessimistic approach in figure 8 is observed to lead a significantly larger, but safe controller. All presented controllers have been generated using the Supremica tool, developed by Åkesson et al. [17].

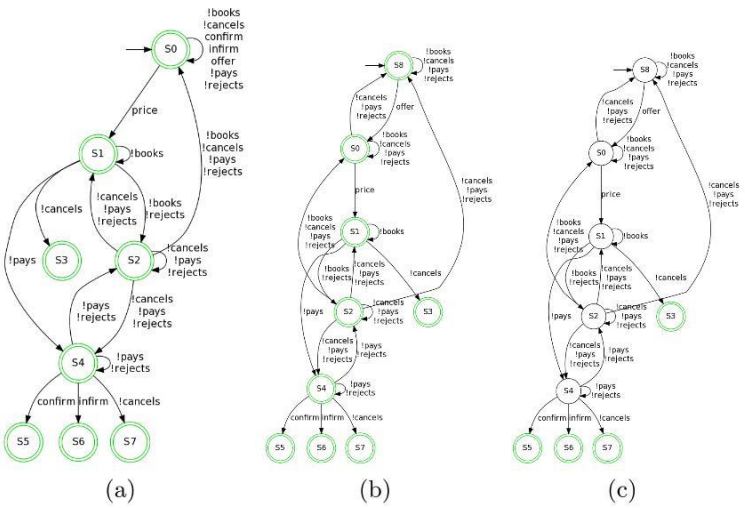


Fig. 5: Refinement process

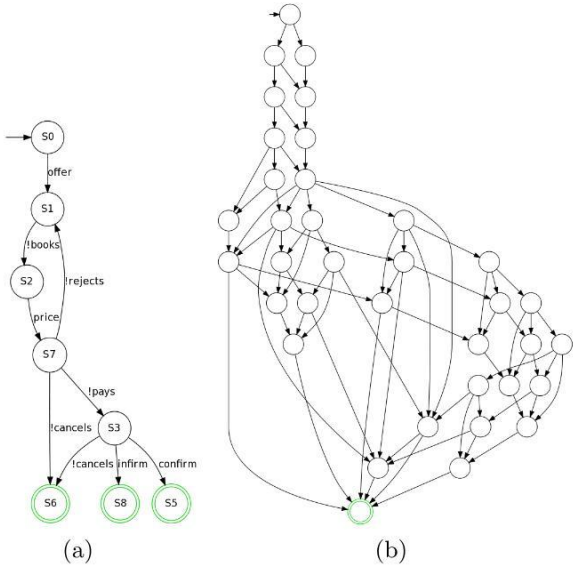


Fig. 6: Optimistic choice (a) U' deterministic refined model and (b) Optimistic controller

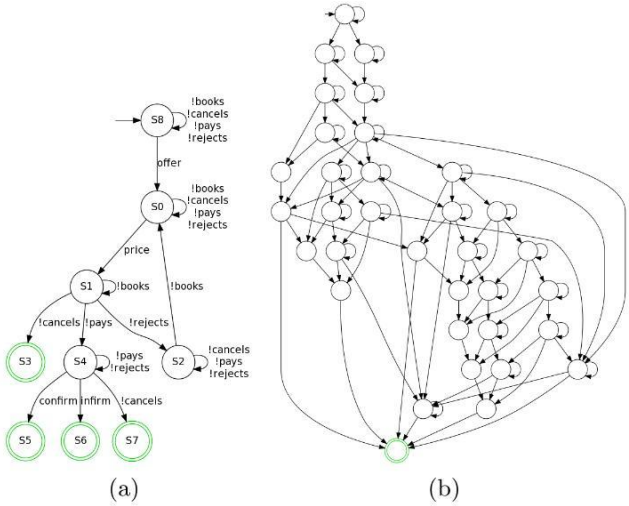


Fig. 7: Semioptimistic choice (a) U' observable refined model and (b) Semioptimistic controller

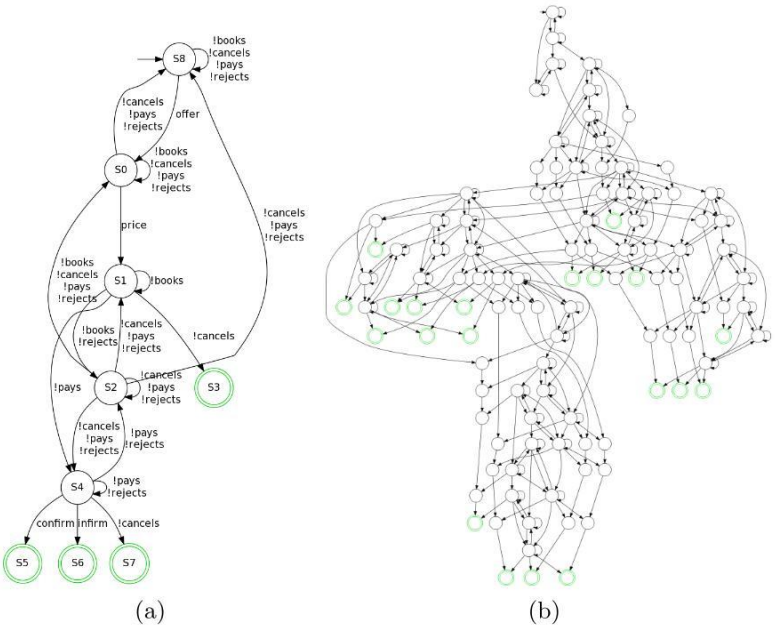


Fig. 8: Pessimistic choice (a) U' nondeterministic refined model and (b) Pessimistic controller

4 Related work

Part of our approach is related to the explaining of counterexamples in [15], as we also explore the vicinity of an observed error trace. However, as it uses model checking for exploration, their work applies only to fully specified programs, while we cannot anticipate the behaviour of the unspecified component and we need to actually run the system.

Another related method is the one by Giannakopoulou et al., which relies on environmental assumption generation when verifying a software component against a certain desired property [14]. Their approach is based on the work of de Alfaro and Henzinger [12] stating that two components are compatible if there exists an environment that enables them to correctly work together. The technique determines if a component satisfies a property for all, some or none possible environments, and it obtains a formal characterisation of the weakest environment needed for the property to hold. By building a correctness-enforcing controller, our approach actually creates an environment in which a desired property would hold. However, while in [14] the analyzed component is well specified, our approach addresses systems that also contain black-box components, whose behaviour and controllability must be understood before building an adaptor.

The use of verification-driven execution also relates our method to the work of Harel et al. in the domain of smart play-out [11, 10]. Smart play-out is a lookahead technique that employs model-checking to execute and analyze Live Sequence Charts. The play-out technique is mainly used to actually execute specifications from a GUI, during the software design process, in order to better understand the application requirements. Both smart play-out and our approach make use of verification-driven execution to improve knowledge, but while we automatically infer the behaviour of an existing, unknown component with the purpose of controlling it, smart play-out experiments with execution scenarios to find the best design options.

Peled et al. have developed a model checking technique for systems in the absence of a given model, or when the model is inaccurate [16, 13]. This approach is closely related to ours, since an approximation for the system model is proposed, verified, and compared to the system behaviour using black-box testing. Found differences are used to generate a new model. When a counterexample is found, it is validated by testing. However, in contrast to our approach, their employed model has only input events, which highly simplifies the learning process, and, also, since the aim is to find feasible counterexamples, their model rather underapproximates the system behaviour. When refining the model, an intermediary learning phase is performed to ensure its consistency, a problem which our algorithm avoids by overapproximating the system behaviour and by allowing for temporary nondeterminism in the tentative model. Also, in our case, the verification, testing and model refinement phases are strongly interleaved, which allows us to obtain a good model earlier, while their approach has distinct phases, since an early confirmed counterexample would save further learning effort. More that just verifying the black-box, our approach explores

correct controllable behaviour in the vicinity of an error trace with the purpose of generating a correctness-enforcing controller.

By starting its analysis from an observed error trace, our approach is also related to Zeller's delta debugging technique [9], which has been applied to the component-based area to isolate sets of interactions relevant to the fault. In this case, the delta debugging algorithm works by recording all the interactions in one erroneous and one correct system execution, and then systematically reproducing parts of the failing and successful scenarios using a specialised tool, JINSI [6]. An important difference with respect to our approach is that delta debugging has a diagnosis-oriented nature, and it doesn't focus on providing a fix for the system. Also, while our solution generates new execution traces, thus inferring new knowledge about the system, JINSI works on minimizing the information in two preexisting logs - thus the two techniques are complementary.

ClearView, developed by Perkins et al. [18], is a tool for automatically patching errors in application software. ClearView learns invariants from observed correct executions, detects failing executions and monitors them to find violations of the former invariants. A set of candidate repair patches are generated, then several instances of the patched applications are observed to select the best patch. Both ClearView and our approach work towards ensuring software robustness: while ClearView only extracts its information out of observing correct executions, our method performs an experimental, directed search of the possible system behaviour space, thus being able to reach corner cases that are usually ignored in normal executions. Also, the ClearView technique is prone to bad invariant-error correlations: only some patches are efficient. By contrast, though not complete, our approach is sound: found error traces, if controllable, are always disabled from further occurrence by the generated controller.

5 Conclusions and future work

We have presented an approach that enables the control of a system containing an unknown component, towards the satisfaction of a desired global property. Starting from a most general model of the unspecified component, our technique explores its runtime behaviour and uses the information acquired to refine the model. The aim is to ensure the avoidance of erroneous executions by building a permissive controller. However, if the system safety requirements demand it, a more restrictive controller can be computed.

Our approach is non-intrusive, thus being well-adapted for legacy software. It addresses the rather frequent problem of integrating insufficiently known components in a system, and resolves observed errors, while improving the component knowledge. Also, the obtained sets of erroneous and correct traces can become maintenance documentation.

We currently work to implement our method and develop a specialised framework for it. For initial experiments, we have used Supremica tool [17] for controller synthesis, while the bounded model-checking and model refinement modules will be locally implemented. We will validate our prototype on a set of

Enterprise JavaBeans component systems using Java Message Service for asynchronous messaging. Future extensions of this work include generating regression tests for the corrected system and synthesizing distributed error-avoiding controllers for remote components.

Acknowledgments We are grateful to Marius Minea and Mihai Balint for consistent and useful feedback. This work was partially supported by the European FP7-ICT-2007-1 project 216471, AVANTSSAR and by the strategic grant POSDRU 6/1.5/S/13, (2008) of the Ministry of Labour, Family and Social Protection, Romania, co-financed by the European Social Fund: Investing in People.

References

1. R. Alur, T. A. Henzinger, and O. Kupferman. Alternating-time temporal logic. *Journal of the ACM*, 2002.
2. A. Arnold, A. Vincent, and I. Walukiewicz. Games for synthesis of controllers with partial observation. *Theor. Comput. Sci.*, 2003.
3. M. Autili, P. Inverardi, A. Navarra, M. Tivoli. Synthesis: A tool for automatically assembling correct and distributed component-based systems. In *ICSE*, 2007.
4. J. Bernet, D. Janin, and I. Walukiewicz. Permissive strategies: from parity games to safety games. In *ITA*, 2002.
5. A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Advances in Computers*, 2003.
6. M. Burger and A. Zeller. Replaying and isolating failing multi-object interactions. In *WODA*, 2008. ACM.
7. C. Canal, P. Poizat, and G. Salaün. Model-based adaptation of behavioral mismatching components. *IEEE Trans. Softw. Eng.*, 2008.
8. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 2003.
9. H. Cleve and A. Zeller. Locating causes of program failures. In *ICSE '05*,ACM.
10. David Harel, S. Maoz. *Concurrency, Compositionality, and Correctness*, chapter On the Power of Play-Out for Scenario-Based Programs, LNCS. Springer, 2010.
11. David Harel, Hillel Kugler, A. Pnueli. Smart play-out. OOPSLA '03, demo paper.
12. L. de Alfaro, T. A. Henzinger. Interface automata. In *ESEC/FSE-9*, 2001. ACM.
13. D. Peled, Moshe Y. Vardi. Black box checking. In *FORTE/PSTV*, Kluwer, 1999.
14. D. Giannakopoulou, C. S. Păsăreanu, and H. Barringer. Component verification with automatically generated assumptions. *Automated Software Engg.*, 2005.
15. A. Groce and W. Visser. What went wrong: Explaining counterexamples. In *SPIN Workshop on Model Checking of Software*, 2003.
16. D. Peled. Model checking and testing combined. In *ICALP*, Springer, 2003.
17. Knut Åkesson, Martin Fabian. Supremica an integrated environment for verification, synthesis and simulation of discrete event systems. In *WODES*, 2006.
18. J. H. Perkins et al. Automatically patching errors in deployed software. In *Proc. of the 21st ACM Symp. on Operat. Syst. Princ.*, 2009.
19. P. Ramadge and W. Wonham. The control of discrete event systems. *Proc. of the IEEE*, 77(1), January 1989.
20. H. W. Schmidt and R. H. Reussner. Generating adapters for concurrent component protocol synchronisation. In *FMOODS*, 2002.
21. D. M. Yellin and R. E. Strom. Protocol specifications and component adaptors. *ACM Transactions on Programming Languages and Syst.*, 1997.

A Formalization of the RTSJ Scoped Memory Model in Dynamic Logic*

Christian Engel and Peter H. Schmitt

Karlsruhe Institute of Technology
Institute for Theoretical Computer Science
D-76128 Karlsruhe, Germany
{engelc,pschmitt}@ira.uka.de

Abstract. The *Real-Time Specification for Java* (RTSJ) features a region-based memory model with the capability to explicitly free memory regions (so called scoped memory areas). For preventing dangling references it introduces runtime checks, raising errors in the case of a failure, to restrict the creation of references between objects residing in different regions. This work presents an operational semantics for the RTSJ memory model formulated in a sequent calculus for dynamic logic. The calculus employs symbolic execution and can be used for proving the absence of failed runtime checks. This is crucial for giving safety guarantees for RTSJ applications or could allow an RTSJ compliant Java Virtual Machine (JVM) to skip this kind of runtime checks resulting in improved performance. The presented approach has been implemented in the KeY system and evaluated for its practical feasibility.

Keywords: real-time, formal verification, formal methods, symbolic execution, dynamic logic, scoped memory.

1 Introduction

In recent years a trend to make Java suitable for safety critical applications could be observed. One of the main deficiencies concerning the suitability of Java for real-time programming is its memory management featuring a garbage collector. For real-time applications it is not acceptable to be interrupted by garbage collection arbitrarily often and for unbounded periods of time since this would lead to indeterministic performance of the application.

The Real-Time Specification for Java (RTSJ) [4] addresses this issue by providing memory areas which can explicitly be freed and which are thus not subject to garbage collection. This introduces of course the danger of dangling references. RTSJ uses runtime checks to prevent the creation of references that can turn into dangling references when a memory area is freed. A failed check raises a

* This research was funded by the EU project DIANA (Distributed equipment Independent environment for Advanced avioNic Applications).

runtime error. This work presents an approach for statically verifying the absence of this kind of runtime errors which is especially relevant when employing RTSJ programs in safety critical systems.

A broader view on the verification of RTSJ programs is given in the dissertation [8] where also examples are provided from which we abstain in this paper due to space restrictions and the paper's focus on the modeling of the RTSJ memory model. Preliminary results on the topics discussed in this paper have been published in the work-in-progress paper [7].

Instead of providing solutions on an abstract level we formulate our approach in a concrete formalism, namely dynamic logic [9], with existing tool support, in this case the KeY system [1] which is a theorem prover featuring a dynamic logic for a sequential subset (excluding for instance reflection) of Java. The operational semantics of RTSJ's scoped memory model formulated as dynamic logic rules in Section 3 could, however, easily be "translated" to other formalisms, e.g. [16]. All presented results were implemented in the KeY system and evaluated for their feasibility. For readers not familiar with dynamic logic a brief introduction is given in Section 2.

The remainder of the paper is organized as follows: Section 2 briefly summarizes the necessary background. For the sake of brevity and comprehensibility we try to keep this part as informal as possible. For a comprehensive account of the theoretical foundations of JAVA DL and the KeY system refer to [3]. Section 3 describes the main contribution of this work, namely the extension of the JAVA DL calculus for treating RTSJ's scoped memory model. Section 4 reviews related work and Section 5 contains a wrap-up of the presented approach and an outlook on future work.

2 Foundations

In the following, the basics of dynamic logic and RTSJ needed to understand this work are briefly summarized.

2.1 Dynamic Logic

First-order dynamic logic (DL) [9] extends first-order predicate logic by a modality $[p]$ for every program p of some imperative programming language. For two first order formulas ϕ and ψ and a legal program $\mathtt{p}$, the formula $\phi \rightarrow [\mathtt{p}]\,\psi$, for instance, is valid iff for every program state s satisfying ϕ the execution of $\mathtt{p}$ when started in s either (i) terminates in a state satisfying ψ or (ii) does not terminate. This matches the semantics of the *Hoare Triple* [11] $\{\phi\}p\{\psi\}$. Beside this example for a partial correctness specification, total correctness is expressible by the diamond modality ($\langle\rangle$). Accordingly, the semantics of $\phi \rightarrow \langle\mathtt{p}\rangle\,\psi$ is that $\mathtt{p}$ terminates when started in an arbitrary state satisfying ϕ and ψ holds in the corresponding post state.

Dynamic logic formulas are interpreted in *Kripke structures*. A Kripke structure is defined by a tuple (S, ρ), where S is the set of first order structures

representing program states and ρ is a function mapping each program p to a *transition relation* $\rho(p) \subseteq S^2$ such that $(s_1, s_2) \in \rho(p)$ iff executing p in s_1 leads to s_2. All states $s \in S$ share the same universe $\mathbf{U}$. Given a state s and a variable assignment β, the evaluation of terms to values (i.e., elements of $\mathbf{U}$) by a function $val_{s,\beta}$ and the validity of first-order formulas $s, \beta \models \varphi$ are defined in the standard way. Symbols that can be interpreted differently in different states (such as program variables) are called flexible, those whose interpretation remains the same in all states (such as predefined arithmetic operators) are called rigid.

The semantics of the underlying programming language, which is needed for performing symbolic execution, is encoded in the calculus rules of a sequent calculus. This calculus operates on proof trees whose nodes are sequents. A sequent $\Gamma \Rightarrow \Delta$, where Γ (the antecedent) and Δ (the succedent) are sets of DL formulas, is valid iff the formula

$$\bigwedge_{\gamma \in \Gamma} \gamma \rightarrow \bigvee_{\delta \in \Delta} \delta \tag{1}$$

is valid. Thus the formulas in the antecedent can also be thought of as assumptions we can use to prove the succedent to be true.

A sequent calculus rule

$$\frac{\Gamma_1 \Rightarrow \Delta_1 \quad \ldots \quad \Gamma_n \Rightarrow \Delta_n}{\Gamma \Rightarrow \Delta} \tag{2}$$

is correct if the validity of the premises (the sequents $\Gamma_i \Rightarrow \Delta_i$ with $1 \leq i \leq n$) implies the validity of the conclusion ($\Gamma \Rightarrow \Delta$). This means that a rule is basically applied bottom up: The conclusion is the sequent the rule is applied to and the premises are the result of the application. For proving the validity of a formula Φ we start with a sequent $\Rightarrow \Phi$.

Since the verification of Java programs has to deal with side effects of program execution resulting in changes of the program state, the dynamic logic for Java utilized in the KeY system, called JAVA DL, provides a means, called updates, to describe those state transitions. An update is basically a lazy substitution which is not evaluated as long as it is applied to a modality. We distinguish elementary, parallel and quantified updates. An elementary update has the form $\{loc := val\}$, where loc and val are terms. The semantics of the update $\{loc := val\}\varphi$ where φ is an arbitrary JAVA DL formula matches the semantics of $\langle$`loc=val;`$\rangle\, \varphi$ iff loc and val are side effect free. Symbolic execution, as we consider it here, computes the effect of a program by compiling it stepwise (operating always on the first statement of the executed program) to an update. For each statement this process is two-phased: First the statement is flattened which means a statement containing complex (i.e. potentially having side-effects) sub-expressions is transformed to an semantically equivalent sequent of simpler statements. Then if no further simplification can be achieved on the first statement it is compiled to an update. A rule for executing an assignment `x=y;` where `x` and `y` are variables is

given by

$$\frac{\Gamma \Rightarrow \{\mathcal{U}\}\{x := y\}\langle \pi\, \omega\rangle\, \varphi,\ \Delta}{\Gamma \Rightarrow \{\mathcal{U}\}\langle \pi\, \texttt{x=y;}\, \omega\rangle\, \varphi,\ \Delta}$$

where $\{\mathcal{U}\}$ is an update, ω represents the remainder of the program following the first executable statement `x=y;` and π a non-executable prefix such as opening braces or *method frames* (introduced in Sect. 3.1).

Besides elementary updates of the form $\{loc := val\}$ we use in the remainder of this work parallel updates of the form $\mathcal{U}_1||\mathcal{U}_2$ (where $\mathcal{U}_1, \mathcal{U}_2$ are updates) and quantified updates $for\, x;\ \phi;\ \mathcal{U}$ (where x is a logic variable bound in ϕ and $\mathcal{U}$, ϕ is a formula and $\mathcal{U}$ is an update). Parallel updates are executed simultaneously (collisions are resolved by a last win semantics), the update $\{x := y||y := x\}$ for instance swaps the values of x and y. The quantified update $\{for\, x;\ 0 \leq x \wedge x < a.length;\ a[x] := null\}$ assigns $null$ to all slots of the array a.

KeY possesses a frontend for the Java Modeling Language (JML) [14] which was used to read JML specifications of the RTSJ API (see Sect. 3). JML specifications are compiled to dynamic logic when imported in KeY.

2.2 Real-Time Specification for Java

The RTSJ [4] takes up the idea of a region-based memory model see e.g., [18]. Two novel kinds of memory regions are added to the classical Java heap memory: *immortal memory* and *scoped memory*. An immortal memory area is never garbage collected and never released during the lifetime of an application. In contrast, a scoped memory area is reclaimed as soon as no thread is active inside it any more. RTSJ does not introduce new syntax. Immortal and scoped memory are represented by ordinary Java classes. Thus, objects representing memory areas can be generated as usual by the `new` operator, with the restriction that *immortal memory* be a singleton class. Associated with every thread is a *current memory area*. A program executing in memory area s_1 may call the `enter` method with an object *logic* of type runnable as a parameter and the scoped memory area, or scope for short, s_2 as receiver. In this case s_1 is called a parent of s_2, or an outer scope of s_2. The code associated with the object *logic* will be executed with current scope s_2. After termination s_1 will again be the current scope. An outer scope can be reentered using the outer scope's `executeInArea` method which resets the current scope but leaves the nesting structure unchanged. This structure may thus be – even for single-threaded programs – a tree, or in the vernacular of RTSJ, a cactus stack. Because of this nesting structure it is also necessary to keep count for every scope of the number of methods active in it. Only if this reference count drops to 0 will the scope be reclaimed.

RTSJ imposes an additional constraint on the scope stack, called the *single parent rule*: each occurrence of a scope s_1 on the cactus stack has the same parent s_2. As a consequence for each occurrence of s_1 the entire branch below s_1 is the same.

An object that is created while the current memory area is s is said to reside in s. When assigning an object `y` to a reference-type location `x.r` or `x[i]` it is

checked at runtime that (i) `y` resides in immortal memory or (ii) `y` resides in a scope s_1 and `x` resides in the same or an inner scope s_2 of s_1. This avoids the situation that y may have been deleted while x is still there. A failed check raises an `IllegalAssignmentError`. If all checks succeed we are sure that no dangling reference occurs.

The following example illustrates how scoped memory can be used:

```
1  public void doWorkInScopedMemory(){
2    ScopedMemory m = new LTMemory(100000);
3    Runnable worker = new Runnable(){ public void run(){
4        ... /*the work to be done in ScopedMemory*/ }};
5    m.enter(worker); }
```

The method `doWorkInScopedMemory` is supposed to perform certain computations in scoped memory: In line 2 a new scope is created, in line 3 and 4 the code (encapsulated in an object of type `Runnable`) to be executed in scoped memory is defined and in line 5 this code is executed in the newly created scope. The objects temporarily allocated in scope `m` are deleted after the invocation of `enter` (line 5) has terminated.

Restrictions We will only be concerned with single threaded programs. Heap memory can also be used by RTSJ applications. In this work we concentrate on the new features and consider only programs running exclusively in immortal and scoped memory.

Furthermore we forbid the reentering of immortal memory using its `enter` method; accessing it via `executeInArea` is permitted however. This ensures that the only place a non-scope memory area can occur on the scope stack is its root. Even though this rules out legal RTSJ programs it is much less restrictive than safety critical Java profiles imposing similar restrictions [17, 12, 10]. In addition we forbid usage of Java threads and only allow threads implementing `javax.realtime.Schedulable`.

3 Operational Semantics

In this section we present an operational semantics of the RTSJ scoped memory model in the form of dynamic logic rules as introduced in Section 2.1. We will focus on the rules required to handle the novel features of RTSJ's semantics on top of the dynamic logic for "standard" Java (JAVA DL) described in [3]. The following extensions will be necessary:

1. Means for keeping track of the current scope and the set of objects allocated in specific scopes (Sections 3.1-3.3).
2. The scope stack is modeled by an abstract Java class whose semantics is described by invariants and calculus rules. The nesting relation of scopes is represented by the partial order $\preceq$ (Sections 3.4 and 3.6).

```
if(self instanceof T1){    v=self.m(v1,...,vn)@(T1);
}else if(self instanceof T2) {    v=self.m(v1,...,vn)@(T2);
}else if(...){...
}else{    v=self.m(v1,...,vn)@(Tm);
}
```

Fig. 1. Case distinction on the dynamic type of `self`

3. The semantics of the relevant parts of the RTSJ API is described by a reference implementation augmented with a formal specification mainly consisting of JML invariants (Section 3.5) and
4. Rules for symbolic execution of illegal assignment checks (Section 3.7).

3.1 Inlining of Method Bodies and the <cma> Pointer

We augment the Java syntax with a pointer `<cma>` (in shorthand for `current memory area`) of type `MemoryArea` which points to the currently active memory area. It is not only similar to a `this` pointer in this respect, it is also technically handled in a comparable way: Execution context information as, for instance, needed for resolving the `this` or `<cma>` pointer is stored in a so-called method-frame statement. Method-frame statements are created when inlining method bodies. To elaborate on this a bit further we now exemplarily consider the symbolic execution of a non-*void* instance method invocation.

As described in Section 2.1 statements containing complex subexpressions are first flattened. For a method invocation this results in a statement of the form (we ignore additional statements introduced by the process of flattening here):

```
T v = self.m(v1,...,vn);
```

where `v`, `self`, `v1` ... `vn` are program variables.

The next step is performing a case distinction (encoded as an `if`-cascade) on the dynamic type (only those subtypes of the static type of `self` implementing `m` have to be considered here) of the receiver object `self`. The `if`-cascade is shown in Figure 1. The statements `v=self.m(v1,...,vn)@(Ti)` are so-called *method-body statements* which serve as placeholders for a concrete method body (in this case for the method body implemented in class `Ti`).

A *method-body statement* is symbolically executed by replacing it with the method body it stands for and enclosing it in a *method frame* (a statement keeping track of the current execution context).

Definition 1 (Method Frame Statement). *Let `r`, `self` and `mem` be program variables, `T` a class type and `p` a sequence of statements then*

```
mf(result->r, source=T, this=self, <cma>=mem) : { p }
```

is a method frame *statement.*

The rule methodBodyExpand replaces a *method-body statement* with the actual method body it stands for:

$$\text{methodBodyExpand}\ \frac{\Gamma \Rightarrow \{\mathcal{U}\}\langle\pi\ \texttt{mf(}\ \texttt{result->res,}\ \texttt{source=T,}\ \texttt{this=self,}\ \texttt{<cma>=mem)}\ :\ \{\texttt{body}\}\ \omega\rangle\phi,\ \Delta}{\Gamma \Rightarrow \{\mathcal{U}\}\langle\pi\ \texttt{res=self.m(args)@(T);}\ \omega\rangle\ \phi,\ \Delta}$$

where *body* is a sequence of assignments assigning the arguments `args` to the formal parameters used in the corresponding implementation of `m` followed by the method body itself. The `<cma>` pointer is set to the value the `<cma>` pointer of the enclosing *method frame* evaluates to at the point the method call occurs which is from a technical point of view just the object `mem` stored in the enclosing *method frame*. Thus the value of `<cma>` is not changed by methodBodyExpand.

As however, on some occasions, such as during the execution of the `enter` or `executeInArea` method, the current scope needs to change we introduce an implicit[1] method `<runRunnable>` to model this circumstance. This method is invoked by the reference implementation of `enter` (see Figure 3) and `executeInArea` when a changeover of the current scope needs to be performed. This behavior of `<runRunnable>` is not encoded in Java code[2] as part of the reference implementation described in Section 3.5 but specified by the following rule instead:

$$\text{expandRR}\ \frac{\Gamma \Rightarrow \{\mathcal{U}\}\langle\pi\ \texttt{mf(}\ \texttt{result->lhs,}\ \texttt{source=MemoryArea,}\ \texttt{this=se,}\ \texttt{<cma>=se)}\ :\ \{\texttt{logic.run();}\}\ \omega\rangle\phi,\ \Delta}{\Gamma \Rightarrow \{\mathcal{U}\}\langle\pi\ \texttt{lhs=rr;}\ \omega\rangle\ \phi,\ \Delta}$$

where `rr` stands for `se.<runRunnable>(logic)@(MemoryArea)`. Consequently rule methodBodyExpand must not be applicable to statements of this form.

3.2 The Implicit Field <ma>

We declare an implicit field <ma> in class object for storing the memory scope the object is allocated in. In our model, every created object is allocated in some memory area. This leads to the rule maNonNull:

$$\text{maNonNull}\ \frac{\Gamma,\ o.\texttt{<c>} \doteq \mathit{TRUE},\ o.\texttt{<ma>} \neq \texttt{null},\ \mathrm{RS} \Rightarrow \Delta}{\Gamma,\ o.\texttt{<c>} \doteq \mathit{TRUE},\ \mathrm{RS} \Rightarrow \Delta}$$

[1] In JAVA DL the terms *implicit field* and *implicit method* refers to fields and methods which do not exist in the original code and are added for providing additional state information or describing certain operations during object creation.

[2] This special handling of the `<runRunnable>` method is mirrored by the fact that the switching of scopes performed by RTSJ-compliant VMs has to be performed by a *native* implementation of the `enter` (resp. `executeInArea`) method since it is not expressible in Java code.

3.3 Object Creation

Having Java as the target language we need to model object creation and initialisation. As many other dynamic logics, and modal logics in general, JAVA DL operates under the technically advantageous constant-domain assumption. This means that all states of a Kripke structure share the same universe which seems contradictory to modeling dynamic object creation since it is not possible to add new elements to the universe.

The implications this carries for the modeling of object creation are that all objects that can ever be created by a program have to exist in *every* program state. Whether an object is created is determined by the values of certain *implicit* fields, when a new object is created these fields have to be changed appropriately. To formalize this approach we introduce the notion of object repositories and repository access functions.

Definition 2 (Object Repository). *Let C be a non-abstract class type. The object repository Rep_C denotes the set of all elements of* $\mathbf{U}$ *of dynamic type C with Rep_C being enumerable.*

Since we defined object repositories to be enumerable sets (which is actually a restriction we imposed on the set of admissible states in JAVA DL Kripke structures) it is possible to index them and provide by this a means to access every repository object including the not yet created ones. This indexing is done by a rigid *repository access function symbol* get_C which evaluates to a surjective mapping $\mathbb{Z} \to Rep_C$ with $I(get_C)|_{\mathbb{N}_0}$ being bijective.

Now as we have a means to talk about all (created and non-created) objects we need to be able to distinguish which objects are already created and which are still "available" when a new instance is to be created. For this we introduce the static field <`nextToCreate`> (which we abbreviate <`ntc`> in the following) for each non-abstract class type C denoting the smallest non-negative index such that the object $get_C(C.\texttt{<ntc>})$ is not created. In addition we require that all objects of dynamic type C having an index greater $C.\texttt{<ntc>}$ are not created either. Thus in each state s the set of created objects is

$$\{s(get_C)(i)|\ i \in \mathbb{Z} \wedge 0 \leq i < val_s(C.\texttt{<ntc>})\}$$

For convenience reasons, we also define a boolean instance field <`created`> (which we abbreviate <`c`> in the following) in `java.lang.Object` which evaluates to *TRUE* if (and only if) the corresponding object is created (We denote the boolean constants with *TRUE* and *FALSE* here to distinguish them from the formulas *true* and $false$).

An instance creation expression is symbolically executed by replacing it by a sequence of calls to implicit methods modeling object creation and initialisation which we do not consider in-depth here. The object allocation itself is represented by the invocation of the implicit method `C.<allocate>` which returns the object $get_C(C.\texttt{<ntc>})$ increases the $C.\texttt{<ntc>}$ pointer and sets the <`c`> attribute of the returned object to *TRUE*. This behavior is encoded in the rule allocate which

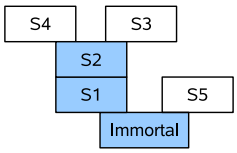


Fig. 2. `S2.stack` represents the substack consisting of `Immortal`, `S1` and `S2`

also initializes the implicit field <ma> introduced in the previous paragraph with the scope `<cma>` refers to at the point `<allocate>` is called:

$$\text{allocate}\ \frac{\begin{array}{l}\Gamma \Rightarrow \{\mathcal{U}\}\{v := get_T(T.\texttt{<ntc>}) \ ||\\ \qquad T.\texttt{<ntc>} := T.\texttt{<ntc>} + 1 \ ||\\ \qquad get_T(T.\texttt{<ntc>}).\texttt{<c>} := \mathit{TRUE} \ ||\\ \qquad get_T(T.\texttt{<ntc>}).\ \texttt{<ma>} := \mathit{scope}\}\langle \pi\, \omega\rangle\, \phi,\ \Delta\end{array}}{\Gamma \Rightarrow \{\mathcal{U}\}\langle \pi\ \texttt{v=T.<allocate>();}\ \omega\rangle\, \phi,\ \Delta}$$

Where *scope* is the object `<cma>` resolves to which is determined by the innermost *method frame* occurring in π.

Tightly coupled with our modeling of object creation is the issue of states reachable by a Java program. Not every state contained in a Kripke structure is also reachable by a Java program. It is however necessary to distinguish reachable from non-reachable states for determining whether axioms known to hold in reachable states can be assumed. Therefore we introduce a flexible predicate RS that holds in exactly those states reachable by a Java program. Properties holding in all reachable states are axiomatised by calculus rules which are applicable only if RS is known to be true (i.e. if it is one of the assumptions given by the antecedent).

3.4 The Scope Stack

We model the scope stack by immutable instances of the Java class `MemoryStack`. Each of these instances only represents a (*local*) sub branch of the entire (global) cactus stack and is as such just a "normal" stack. Each scope is augmented with an attribute `stack` representing the subbranch of the cactus stack ending with the considered memory scope and starting with the immortal memory at the root of the stack. This is possible, as for every scope s the branch below each occurrence of s on the cactus stack is identical. If `stack` is `null` the scope is not located on the cactus stack.

Whenever a new scope that is not yet located on the cactus stack is added to it (by execution of the scope's `enter` method) its stack is initialised with the stack created by pushing the newly entered scope on the stack of the currently active memory area (the memory area from which it was entered). By doing this the local stack of the currently active memory area (as well as the stacks of all other scopes on the global stack) is not changed since the instances of `MemoryStack` are immutable.

The Substack Relation $\preceq$ We now need a specification for class `MemoryStack` that is sufficiently expressive but also simple enough to be efficiently used in a verification system. Simplicity of our formalization of `MemoryStack` is crucial since we do not only need to reason over the structure of the scope stack when we call methods on an instance of `MemoryArea` but each time we perform an assignment to an instance attribute or an array slot. A lightweight formalization of the behavior of the scope stack achieving the required simplicity is to define a substack relation $\preceq$, where $a \preceq b$ is true for two local stacks a and b if a is a prefix of b. An alternative formalization of the scope stack and its drawbacks compared to the one proposed here are discussed in [8]. For creating new instances of `MemoryStack` we define an instance method (of class `MemoryStack`) `push` which returns a newly created instance of type `MemoryStack` such that `s` $\preceq$ `s.push(a)` holds for every stack `s` and every scope `a`. Due to the immutability of `MemoryStack`, $\preceq$ is rigid (its arguments, however, can be flexible) which makes it less involved to handle in dynamic logic than it would be the case for a flexible symbol. However, the `stack` attribute of a scope can be set to a different value during the program run. This means that each instance of `MemoryStack` is only a snapshot of a part of the scope stack taken at a certain time. Given a state s, a variable assignment β and two terms a and b with of type `MemoryArea`, a represents an outer scope of b in state s if and only if $s, \beta \models a.stack \preceq b.stack$.

As described in Sect. 3.3, JAVA DL operates under the constant domain semantics which means all instances that can ever be created by the program have to exist from the beginning. Since $\preceq$ is rigid the createdness of objects does not influence its evaluation and it thus has also to be defined on not yet created objects. To make sure that we can always create a stack of which the current stack is a substack we have to require that for every integer i there are infinitely many j with $j > i$ and $get_{MemoryStack}(i) \preceq get_{MemoryStack}(j)$. We encode this behavior of the scope stack in the rule *push*:

$$\text{push}\ \frac{\begin{array}{l}\Gamma,\ \{\mathcal{U}\}\,(b \preceq get_{MemoryStack}(j) \land j \geq MemoryStack.\texttt{<ntc>}) \Rightarrow \\ \{\mathcal{U}\}\{\ a := get_{MemoryStack}(j)\ \|\| \\ \qquad MemoryStack.\texttt{<ntc>} := j + 1\ \|\| \\ \qquad for\ i;\ MemoryStack.\texttt{<ntc>} \leq i \land i \leq j; \\ \qquad\quad get_{MemoryStack}(i).\texttt{<c>} := TRUE\ \|\| \\ \qquad get_{MemoryStack}(j).\ \texttt{<ma>} := b.\ \texttt{<ma>}\}\langle \pi\, \omega \rangle\, \phi, \Delta\end{array}}{\Gamma \Rightarrow \{\mathcal{U}\}\langle \pi\ \texttt{a=b.push(s);}\ \omega \rangle\, \phi,\ \Delta}$$

where j is a fresh (i. e. not yet occurring in the regarded sequent) integer constant. By adding the assumption that $j \geq MemoryStack.\texttt{<ntc>}$ and setting a to $get_{MemoryStack}(j)$, we can express that the result of `push` is some newly created `MemoryStack`. Note, that we cannot just simply set b to

$$get_{MemoryStack}(MemoryStack.\texttt{<ntc>})$$

since then we could deduce from this rule by induction that

$$\begin{array}{l}\forall\, int\ x, y; (x \geq MemoryStack.\texttt{<ntc>} \land y \geq x) \rightarrow \\ get_{MemoryStack}(x) \preceq get_{MemoryStack}(y)\end{array}$$

which is obviously false. Also note, that a local stack does not have a reference to the scope it is associated with, only the corresponding scope possesses a reference to its local stack which is established by the code shown in Figure 3. This is also the reason why the scope `s` which represents the argument of the `push` method in the conclusion of rule push does not occur in the rule's premise.

The Predicate im For distinguishing the scope stack of the immortal memory area from other scope stacks we introduce the predicate im with $\mathsf{im}(s)$ evaluating to *true* if and only if s is the scope stack of the immortal memory area.

3.5 Specification of `MemoryArea` and its Sub-Types

The behaviour of memory areas is described by a *reference implementation* of class `MemoryArea` and its subtypes augmented with class invariants expressed in JML. This implementation can be symbolically executed by the JAVA DL calculus during proof search. An alternative way of formalizing the RTSJ API would be via calculus rules which would, however, result in a formalization that is (i) less (human-) readable and (ii) less suitable for cross-verification of the formalization's correctness which could, for instance, be performed against a JML formalization of the (natural language) specification provided by RTSJ. Due to the restrictions (see Sect. 2.2) imposed on the memory model all memory areas we permitted (immortal and scoped memory) behave basically like scoped memory areas: since calling the `enter` method on the immortal memory area is no longer allowed its behavior is basically identical to that of a scoped memory area. The mere difference of the immortal memory compared to "normal" scopes is its distinct position at the bottom of the scope stack. Due to this uniformity it is sufficient to provide a reference implementation for `javax.realtime.MemoryArea`: the classes `ScopedMemory`, `LTMemory` and `VTMemory` do not need to override any of the `enter` and `executeInArea` methods since their behaviour is sufficiently described by the implementation and specification of `MemoryArea`. Only for `ImmortalMemory` we have to strengthen (compared to `MemoryArea`) its class invariants for the afore mentioned reasons. Figure 3 shows the implementation of method `enter` in `javax.realtime.MemoryArea`.

3.6 Axiomatization of $\preceq$ and im

By what has been described so far $\preceq$ is merely an uninterpreted predicate, so we have to axiomatize what it should semantically stand for. This is done via calculus rules as exemplarily (due to space constraints it is not possible to show all rules here) demonstrated in the following. The relation $\preceq$ is a partial order and thus reflexive, transitive and antisymmetric. Reflexivity for instance, is axiomatized by rule outerScopeReflexive

$$\mathsf{outerScopeReflexive}\ \frac{\Gamma,\ o \neq \texttt{null} \Rightarrow \mathit{true},\ \Delta}{\Gamma,\ o \neq \texttt{null} \Rightarrow o \preceq o,\ \Delta}$$

```
public abstract class MemoryArea{
  ...
  public void enter(java.lang.Runnable logic){
    if(logic==null) throw new IllegalArgumentException();
    if(parent!=null && parent!=<cma>) throw new ScopedCycleException();
    parent = <cma>;
    referenceCount++;
    if(stack==null) stack = <cma>.stack.push(this);
    try{
      <runRunnable>(logic);
    }catch(Exception e){
        if(this==getMemoryArea(e)) throw RealtimeSystem.tbe();
    }finally{
      referenceCount--;
      if(referenceCount==0){
        consumed=0; parent=null; stack=null;
      }
    }
  }
```

Fig. 3. The `enter` method implemented in class `MemoryArea`

The rule outerRefAttr1 states that in every reachable program state all non-static attributes which are (i) different from `Object.<ma>`, `MemoryArea.parent` and `MemoryArea.logic` and (ii) not `null` point to the same or an outer scope:

$$\text{outerRefAttr1}\ \frac{\Gamma,\ o.\texttt{<c>} = \mathit{TRUE},\ o.a \neq \texttt{null},\\ o.a.\texttt{<ma>.stack} \preceq o.\texttt{<ma>.stack},\ \mathrm{RS}\ \Rightarrow \Delta}{\Gamma,\ o.\texttt{<c>} = \mathit{TRUE},\ o.a \neq \texttt{null},\ \mathrm{RS} \Rightarrow \Delta}$$

A corresponding rule for arrays expresses that objects referenced by array slots are allocated in the scope containing the array itself or an outer scope of it.

Immortal memory always occurs as a singleton. Since the stack associated with this immortal scope does not change either during the program run we can assume that there can be only one stack for which im holds:

$$\text{imUnique}\ \frac{\Gamma,\ \mathsf{im}(o_1),\ \mathsf{im}(o_2),\ o_1 \doteq o_2 \Rightarrow \Delta}{\Gamma,\ \mathsf{im}(o_1),\ \mathsf{im}(o_2) \Rightarrow \Delta}$$

The immortal scope is the outermost scope on the scope stack. Thus the only stack that is a sub stack of the stack belonging to the immortal scope is the stack of the immortal scope itself:

$$\text{imSub1}\ \frac{\Gamma,\ o_1 \preceq o_2,\ \mathsf{im}(o_2),\ o_1 \doteq o_2 \Rightarrow \Delta}{\Gamma,\ o_1 \preceq o_2,\ \mathsf{im}(o_2) \Rightarrow \Delta}$$

3.7 Rules for Symbolic Execution

For the evaluation of RTSJ programs by symbolic execution we have to take into account the runtime checks an RTSJ compliant JVM performs and the exceptions originating from this. Luckily, this applies only to `IllegalAssignmentErrors` since all other RTSJ typical exceptions we deal with are handled by the reference implementation of the RTSJ memory management API. In the following the symbolic execution rules for assignment operations necessitating such illegal-assignment runtime checks are described. These assignments can be subdivided in two classes: (i) assignments to static references and (ii) assignments to non-static references (instance attributes, array slots). In the former case the assigned object has to reside in immortal memory, in the latter case it is sufficient that it resides in the same or in a more outer scope than the object (array) whose attribute (array slot) constitutes the left-hand side of the assignment.

Assignments to Static References Static reference type attributes can only reference `null` or objects allocated in immortal memory. This has to be checked when assigning an object to a static attribute (sa denotes a reference type static attribute and v a program variable of compatible type):

$$\text{sRAttrWrite}\ \frac{\begin{array}{l}\Gamma,\ \{\mathcal{U}\}(\mathsf{im}(v.\texttt{<ma>.stack}) \lor v \doteq \texttt{null}) \Rightarrow \{\mathcal{U}\}\{sa := v\}\langle\pi\,\omega\rangle\,\varphi,\ \Delta \\ \Gamma,\ \{\mathcal{U}\}(\neg\,\mathsf{im}(v.\texttt{<ma>.stack}) \land v \neq \texttt{null}) \Rightarrow \{\mathcal{U}\}\langle\pi\,\texttt{IAE;}\,\omega\rangle\,\varphi,\ \Delta\end{array}}{\Gamma \Rightarrow \{\mathcal{U}\}\langle\pi\,\texttt{sa=v;}\,\omega\rangle\,\varphi,\ \Delta}$$

In the above rule we distinguish two cases (indicated by the two premises): Either the assignment is legal since v is `null` or allocated in immortal memory. In this case the assignment is executed an the resulting state change is expressed by the update $\{sa := v\}$. Or the assignment is illegal since v is not `null` and not allocated in immortal memory. This leads to an `IllegalAssignmentError` to be raised. The statement `throw new IllegalAssignmentError();` was abbreviated with `IAE;` in the above rule.

Assignments to Non-Static References For write accesses to instance attributes we have to distinguish two cases: Either the attribute assigned to is `parent@(MemoryArea)` or it is not. In the former case no check for an illegal assignment is performed since the attribute `parent` may actually refer to objects in inner scopes. As `parent` is only used for modeling purposes and does not correspond to a real attribute in `MemoryArea` this treatment cannot lead to undetected `IllegalAssignementErrors`. In the latter case we have to take into account the possibility that the assignment raises an `IllegalAssignementError` which is modeled by the following rule (where o and v are reference type program variables and a is a reference type attribute different from `parent@MemoryArea`):

$$\text{rAttrWrite}\ \frac{\begin{array}{l}\Gamma,\ \{\mathcal{U}\}(o \neq \texttt{null} \land \psi) \Rightarrow \{\mathcal{U}\}\{o.a := v\}\langle\pi\,\omega\rangle\,\varphi,\ \Delta \\ \Gamma,\ \{\mathcal{U}\}o \doteq \texttt{null} \Rightarrow \{\mathcal{U}\}\langle\pi\,\texttt{NPE;}\,\omega\rangle\,\varphi,\ \Delta \\ \Gamma,\ \{\mathcal{U}\}(o \neq \texttt{null} \land \neg\psi) \Rightarrow \{\mathcal{U}\}\langle\pi\,\texttt{IAE;}\,\omega\rangle\,\varphi,\ \Delta\end{array}}{\Gamma \Rightarrow \{\mathcal{U}\}\langle\pi\,\texttt{o.a=v;}\,\omega\rangle\,\varphi,\ \Delta}$$

where $\psi := v \doteq \texttt{null} \vee v.\texttt{<ma>.stack} \preceq o.\texttt{<ma>.stack}$ states that v is either `null` or allocated in the same scope as o or an outer scope of it. In the above rule we have to distinguish three cases (indicated by the three premises): (i) o is not `null` in the state described by the update $\mathcal{U}$ and the assignment is legal with respect to the RTSJ constraints on this (i.e. when formula ψ holds), (ii) $o \doteq \texttt{null}$ holds and a `NullPointerException` is raised (abbreviated by `NPE;`) or (iii) the former two case do not hold and an `IllegalAssignmentError` is thrown (abbreviated by `IAE;`).

Assignments to attributes without an explicit prefix (i.e. being prefixed with an implicit `this` reference) are handled analogously after determining the receiver object on the left-hand side from the execution context π. Assignments to array slots are handled in an analogous way to assignments to instance attributes.

3.8 Example

We now attempt to verify that method `doWorkInScopedMemory` (as shown in Section 2.2) always terminates without raising an exception (that is not caught within the considered method itself). This is expressed by the proof obligation:

$$\Rightarrow \psi \rightarrow \langle \texttt{doWorkInScopedMemory();} \rangle \mathit{true} \tag{3}$$

Where ψ is a precondition consisting of some implicit assumptions and class invariants assumed to hold in every pre-state of `doWorkInScopedMemory`.

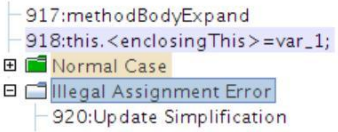


Fig. 4. Snipped of a proof tree as presented by the KeY system

Attempting to verify proof obligation (3) with the KeY system fails. A look on the proof tree (see Figure 4) reveals the reason for this: Instances of anonymous classes (in our example the `Runnable` object created by `doWorkInScopedMemory`) hold a reference (represented by the implicit field `<enclosingThis>`) to the object the `this` pointer resolves to in the context the instance of the anonymous class was created. Creating this reference can lead to an `IllegalAssignmentError` if the scope containing the `this` object is not an outer scope of the scope containing the instance of the anonymous class. Strengthening the precondition by adding the formula

$$\mathit{self}.\texttt{<ma>.stack} \preceq \mathit{defaultMemoryArea}$$

where $\mathit{defaultMemoryArea}$ denotes the scope in which `doWorkInScopedMemory` is executed, solves this problem.

4 Related Work

The presented work focuses on verifying real-time Java programs complying (with only minor restrictions; see Sect. 2.2) with the existing RTSJ. Most related approaches try to improve analyzability of real-time Java applications by further restricting or changing the RTSJ memory model losing, in turn, some of the flexibility it provides.

Kwon and Wellings [13] describe a memory management model making use of implicitly created memory scopes associated with each method leading to something comparable to stack allocation of objects only locally used by a method. The absence of explicit scope identities, for instance, eliminates the need for enforcing the single parent rule since it is impossible to reenter a scope. `IllegalAssignemntErrors` still remain an issue to consider, but checking their absence statically is eased by the simpler memory model and can for instance be done by escape analysis [6].

The application of the software model checker Java PathFinder to RTSJ programs is described in [15]. The approach is based on a reference implementation of the RTSJ API but as PathFinder's JVM was not adapted, scoped memory as well as runtime checks for illegal assignments could not be modeled.

To reduce the error-proneness of RTSJ programs several profiles for safety critical Java (SCJ) applications have been proposed [12, 17, 10] building upon RTSJ and imposing restrictions, for instance, on the nesting hierarchy of scopes.

Several works [2, 5, 19] have proposed an encoding of the nesting relation of scopes in the type system. The outlives relation between memory regions defined in [5] bears similarities to the relation $\succeq$ used in Sect. 3. One major difference is however that, unlike in [5], $\succeq$ represents only a snapshot of the nesting relation between scopes, thus allowing it to change during the program run.

5 Conclusion and Future Work

This paper presented a formalization of RTSJ's memory model facilitating formal verification of real-time Java programs. This does, however, not eliminate the need for SCJ profiles or programming guidelines constraining the use of scoped memory, since every sensible restriction imposed on it might ease its verifiability.

As usual, the correctness of the employed formalization is an important issue to consider here. Most of the defined calculus rules as well as the reference implementation are a rather canonical representation of the semantics provided by the natural language specification RTSJ. However, as part of future work one could think of a cross-verification of the presented formalization against other formal specifications of RTSJ. This could, for instance, incorporate the formal verification of the used reference implementation against a formal JML specification (which would need to be derived from the existing natural language specification) of the corresponding parts of the RTSJ API.

The approach presented in this paper was implemented in the KeY system and successfully tested on several non-trivial examples, some of which are provided in [8].

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(1):32–54, 2005.
2. C. Andreae, Y. Coady, C. Gibbs, J. Noble, J. Vitek, and T. Zhao. Scoped types and aspects for rt Java memory management. *Real-Time Syst.*, 37(1):1–44, 2007.
3. B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer, 2007.
4. G. Bollella and J. Gosling. The RT spec for Java. *Computer*, 33(6):47–54, 2000.
5. C. Boyapati, A. Salcianu, W. Beebee, and J. Rinard. Ownership types for safe region-based memory management in real-time Java. *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2003.
6. J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff. Escape analysis for Java. *SIGPLAN Not.*, 34(10):1–19, 1999.
7. C. Engel. Deductive Verification of RTSJ Programs. In *Proceedings of the 2nd Junior Researcher Workshop on Real-Time Computing (JRWRTC 2008)*, 2008.
8. C. Engel. *Deductive Verification of Safety-Critical Java Programs*. PhD thesis, Karlsruhe Institute of Technology, 2009.
9. D. Harel. Dynamic logic. In D. Gabbay and F. Guenther, editors, *Handbook of Philosophical Logic*, volume 2 of *Extensions of Classical Logic*, pages 497–604. D. Reidel Publishing Company, 1984.
10. HIJA. High Integrity Java Applications. Website: http://www.hija.info, 2006.
11. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.
12. J. Kwon, A. Wellings, and S. King. Ravenscar-Java: a high-integrity profile for real-time Java. *Concurr. Comput. : Pract. Exper.*, 17(5-6):681–713, 2005.
13. J. Kwon and A. J. Wellings. Memory management based on method invocation in RTSJ. In *OTM Workshops*, pages 333–345, 2004.
14. G. T. Leavens, A. L. Baker, and C. Ruby. Prelim. design of JML: a behavioral interface specification language for Java. *SIGSOFT Softw.Eng.Notes*, 31:1–38, 2006.
15. G. Lindstrom, P. C. Mehlitz, and W. Visser. Model checking real time java using java pathfinder. In D. Peled and Y.-K. Tsay, editors, *ATVA*, volume 3707 of *Lecture Notes in Computer Science*, pages 444–456. Springer, 2005.
16. G. D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.
17. M. Schoeberl, H. Sondergaard, B. Thomsen, and A. P. Ravn. A profile for safety critical Java. In *ISORC '07: Proceedings of the 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, pages 94–101, Washington, DC, USA, 2007. IEEE Computer Society.
18. M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 1997.
19. T. Zhao, J. Noble, and J. Vitek. Scoped types for real-time Java. In *RTSS '04: Proceedings of the 25th IEEE International Real-Time Systems Symposium*, pages 241–251, Washington, DC, USA, 2004. IEEE Computer Society.

Specifying Imperative ML-like Programs Using Dynamic Logic *

Séverine Maingaud[1], Vincent Balat[1], Richard Bubel[2], Reiner Hähnle[2], and Alexandre Miquel[3]

[1] Laboratoire *Preuves, Programmes et Systèmes*
CNRS and Université Paris Diderot – Paris 7
[2] Department of Computer Science and Engineering
Chalmers University, Gothenburg
[3] ENS Lyon, Université de Lyon,
LIP (UMR 5668 CNRS ENS Lyon UCBL INRIA)

Abstract. We present a logical system suited for specification and verification of imperative ML programs. The specification language combines dynamic logic (DL), explicit state updates and second-order functional arithmetic. Its proof system is based on a Gentzen-style sequent calculus (adapted to modal logic) with facilities for symbolic evaluation. We illustrate the system with some example, and give a full Kripke-style semantics in order to prove its correctness.

Key words: ML, dynamic logic, program specification, program verification, KeY, AF2

1 Introduction

We present a logical system suited for specification and verification of imperative ML programs. Verification systems for functional programming languages have been traditionally investigated in the context of higher-order logical frameworks (e.g., Coq, Isabelle, HOL, ACL2, VeriFun, Elf), where structural induction is the central proof paradigm. To employ dynamic logic and symbolic execution constitutes a new departure which is motivated by the presence of reference types whose treatment is well understood in Hoare-style program logics. In our paper we show that dynamic logic is a suitable framework also for ML with references. Our specification language combines a generalisation of Hoare logics called dynamic logic (DL), explicit state updates, and second-order functional arithmetic (AF2) [9]. Its proof system is based on a Gentzen-style sequent calculus (adapted to modal logic) with facilities for symbolic evaluation.

ML with references is a higher-order imperative programming language that can be seen as an object-oriented language. Functions and references can be

* This work has partially been supported by the EU COST Action IC0701: Formal Verification of Object-Oriented Software.

translated by objects[4]. For this reason, the work presented in this paper could be adapted to a real object-oriented programming language.

Related Work. State-of-the-art verification systems based on dynamic logic are KIV [1] and KeY [3]. The idea of using updates to represent state changes in a dynamic logic setting originated also from KeY. We depart from KeY's program logic, however, in two main aspects: (i) we use second-order dynamic logic to be able to deal with a functional language, thus bridging the gap between DL and AF2; (ii) memory allocation extends the domain of the store in contrast to the constant-domain assumption employed by KeY.

The proof assistant PAF! [2] is also a verification system for ML programs based on AF2 with symbolic evaluation, but it does not support verification of imperative ML. The verification tool WHY [5] for first-order imperative programs is a verification condition generator based on Dijkstra's weakest precondition calculus. WHY is being adapted to higher-order programs [8] through the integration of effect polymorphism to previous work [10] on Hoare logics for call-by-value functional programs without states. In this setting the generated verification conditions are passed on to automatic theorem provers such as SMT solvers or to interactive proof systems like Coq or PVS. The Ynot system [4] uses Coq both as a theorem prover and as an imperative functional language thanks to a monadic formulation of separation logic.

2 Dynamic Logic

Dynamic logic [6] can be seen as a class of modal logics suited for reasoning about imperative programs. Like Hoare logic it uses a specification language where the current program state is implicit. States are explicit only in the semantics, where they play the role of worlds of a Kripke frame, in the sense of modal logic.

The central idea is to introduce for each program p a separate modality (read 'box p') $[\,p\,]$ whose accessibility relation in a Kripke frame corresponds exactly to the operational semantics of p: the formula $[\,p\,]\,B$ holds in a state s if the formula B holds in all states reachable by any execution of the program p. If p is deterministic (which we assume from now on) then there is at most one final state. Under this semantics the formula

$$A \to [\,p\,]\,B \tag{1}$$

expresses *partial correctness* of program p with respect to precondition A and postcondition B. Whenever A and B are first-order formulas (1) corresponds to the *Hoare triple* $\{A\}\,p\,\{B\}$ [7]. In contrast to Hoare logic, however, in dynamic logic modal operators with programs inside and propositional connectives can be arbitrarily nested which makes dynamic logic more expressive than Hoare logic. In addition to the partial correctness modality there is a dual operator (read "diamond p") $\langle\,p\,\rangle$ defined as $\langle\,p\,\rangle\,B \leftrightarrow \neg[\,p\,]\,\neg B$. Using the diamond operator we can express *total correctness* of program p in dynamic logic: $A \to \langle\,p\,\rangle\,B$.

[4] with a field for any argument and a unique method of application in the first case ; one field representing to the content and two methods get and set in the second case.

In contrast to higher-order logics, imperative programs are first-class citizens in dynamic logic and not modelled by (inductively defined) formulas. In consequence, the syntax and semantics of the underlying programs is fixed and one must define a specific dynamic logic for a given programming language. One advantage is that the programming language semantics is defined at the meta-level (as a property of Kripke frames) and needs not to be defined on the formula level. Likewise, programs can have any concrete syntax and need not follow a formula structure. This leads to a low formalization overhead and good readability when constructing proof obligations for program correctness which in turn is important for (i) handling complex target languages, (ii) achieving a high degree of automation, (iii) usability in interactive proofs.

Proof systems for dynamic logic do not proceed mainly via induction over the syntactic structure of programs, but by decomposition of programs and recording of intermediate (symbolic) states. If the application of decomposition rules follows the evaluation strategy of an interpreter of the underlying programming language, then this amounts to *symbolic execution*. The program-free part of dynamic logic is usually a standard first-order logic with sorts and interpreted symbols for arithmetic, arrays, etc. There is relatively strong automated reasoning support available for such logics. Two state-of-art software verification systems (KeY [3] and KIV [1]) with a very high degree of automation are based on dynamic logic and symbolic execution.

Functional Programs with References. Our programming language is an untyped version of *imperative ML* (IML). Imperative ML adds references (locations) with mutable content to the functional world. References are pointers to a fixed memory location. The value stored at that particular location can be accessed and changed by programs. Let r denote a reference: The IML fragment

$$r := 3; !r$$

consists of two (sequentially connected) expressions: the first expression $r := 3$ changes the content stored at the memory location referred to by r to 3; the second expression $!r$ looks up and evaluates to the value stored in r. Expressions composed by the semicolon operator are evaluated from left-to-right. The resulting value is the one of the last expression; the above IML fragment evaluates always to 3. More details are in Sect. 3.

When extending a functional language with references (and thus with a notion of state) one has to deal with phenomena such as side-effects, aliasing, or sensitivity to evaluation order for functional correctness. For instance, the IML λ-expression

$$f := \lambda x, y.\ ((x := !x + 2; !x) + (y := !y * 5; !y))$$

(applied to arguments) has not only global visible side-effects (contents of references passed to x, y changed), but is also affected by aliasing and the evaluation order: let r, s denote *distinct* references with *equal* content (say 3), then $(!f)\ r\ s$ evaluates to 20 and $(!f)\ r\ r$ evaluates to 30 (under left-to-right evaluation).

The specification language (logic) for IML programs needs not only to model the additional concepts faithfully, but must also ensure that the properties to be specified are actually expressible. For example, in a pure functional setting the

formula $\forall x.(f\ x \leq g\ x)$ specifies that function g is an upper approximation of the program (function) f, but more thought is required in presence of side-effects where executing f might influence the evaluation of g.

Dynamic Logic. We sketch the basic concepts and ideas behind dynamic logic. A rigorous introduction of second-order dynamic logic for IML program is given in Sect. 4. Signature and syntax of dynamic logic are defined on top of an existing non-modal base logic (e.g., first-order or second-order logic). An important feature of first-order modal logics is the distinction between rigid and non-rigid function/predicate symbols. Rigid symbols are interpreted independent of a state, while the interpretation of non-rigid symbols is state-dependent. For instance, the interpretation of the IML dereferencing operator ! must obviously be state-dependent.

The inductive definition of DL syntax is fairly standard. Any formula of the underlying non-modal base logic is also a formula of its dynamic logic variant. Modalities are added to the syntax as follows: let p be an IML program, ϕ denote a DL formula then $[\,p\,]\,\phi$ and $\langle\,p\,\rangle\,\phi$ are DL formulas. An important restriction is that ML programs occuring as logical terms (i.e., outside a modality) must be state-independent and pure (side-effect free).

States in dynamic logic are not represented by an explicit datastructure passed as an extra argument to functions (predicates), but live solely on the semantical level. Formulas and terms are evaluated relative to a *Kripke structure* $\mathcal{K}$. Besides the elementary data domain and an interpretation for the rigid symbols the Kripke structure fixes also a set of states St, giving meaning to non-rigid symbols such as !, and a state transition relation $\tau : \Pi_{IML} \times St \times St$ that defines the semantics of IML programs. The cardinality of $\tau(\pi, s) = \{s' \mid \tau(\pi, s, s')\}$ is at most 1, because IML is deterministic.

Example 1. The DL formula

$$[\,\mathsf{if}\ a > b\ \mathsf{then}\ max := a\ \mathsf{else}\ max := b\,]\,([!max\ \ \mathsf{as}\ x]\,x \geq a)$$

specifies that if the program inside the first box modality terminates then in the final state the value stored at max is at least as large as the value of a. The construct "$\mathsf{as}\ x$", introduced in Sect. 4, is a binder to recover the returned value.

Proof systems for DL typically use a sequent style calculus and follow the symbolic evaluation paradigma by realising a symbolic interpreter. The rule that handles assignment is often one of the most tricky ones and crucial for the efficiency of the verification process. Even for simple imperative languages the standard assignment rule requires renaming of locations and, in presence of aliasing, the introduction of several case distinctions. The update mechanism sketched in the following provides an elegant way to deal with this.

Update Mechanism. Influenced by *abstract state machines* and *generalized substitutions* (B method), the KeY verification system [3] introduced updates as a syntactical notion to represent symbolic state changes in dynamic logic.

An (elementary) *update* is an expression of the form $location := value$. By sequential composition of updates $u_1; u_2$ new (sequential) updates can be built.

More complex update combinators are described in [11], but for our purposes elementary updates plus sequential composition is sufficient.

Let ξ denote a formula or term and u an update: then $\{u\}\xi$ is again a formula/term. The semantics of an elementary update is that of an assignment. In this paper we restrict the kind of term that may occur as location or as an assigned value to so-called *symbolic values*. Simply expressed, a symbolic value is a logical term or a program that has no side-effects and that is not state-dependent.

Example 2.
1. In the formula $\{l := v\}\phi$, the subformula ϕ is evaluated in a state where $!l$ has the value v.
2. The formula $\{l := v1; l := v2\}\phi$ is equivalent to $\{l := v2\}\phi$, because the second update overwrites the effect of the first one.
3. The update in $\{l1 := 3; l2 :=!l1\}\phi$ is syntactically incorrect as the right side of the second update is state dependent and not a symbolic value according to the definition above.

During a sequent proof the updates accumulate in front of a symbolically executed program until execution terminates. Upon termination, the updates are applied to terms and formulas much like substitutions. This lazy application of updates helps efficiency, because automatic first-order simplification steps are applied eagerly *before* updates are substituted into formulas. This is particularly important in presence of aliasing, see Sect. 4.

3 Programming Language

We present the syntax and evaluation rules of a small untyped functional language with references. In this framework, we consider static typing as an attribute of the logic, and we ignore it when defining the operational semantics. Typing can be introduced in the logic as predicates using the power of second order. This allows to reason on programs independently from any typing account.

3.1 Syntax

Constants The language provides two kinds of constants: *integer constants* $n \in \mathbb{Z}$ and *location constants* $\ell \in \mathcal{L}$, where $\mathcal{L}$ is an infinite set of symbols disjoint from $\mathbb{Z}$. Boolean values 'true' and 'false' are represented by the integer constants 1 and 0. In conditionals, we shall more generally consider that any value different from 0 represents the Boolean value 'true'.

Primitive functions We assume a finite set of function symbols (notation: f, f', f_1, etc.) representing elementary operations on data. Every function symbol f comes with an arity $k \geq 1$ and a total function $\tilde{\mathsf{f}} : \mathbb{Z}^k \to \mathbb{Z}$ defining the corresponding operation. We assume that these primitives contain at least the usual arithmetic operations ($+$, $-$, $*$, $/$, etc).

Programs and values The syntactic category of *programs* (notation: p, p', p_1, etc.) is defined by

$$\begin{array}{rcl} p & ::= & x \mid n \mid \ell \mid \mathsf{f}(p_1, \ldots, p_n) \mid p = p' \\ & \mid & \lambda x.\, p \mid p\, p' \mid (p_1,\, p_2) \mid \mathsf{fst}(p) \mid \mathsf{snd}(p) \\ & \mid & \mathsf{if}\ p\ \mathsf{then}\ p_1\ \mathsf{else}\ p_2 \mid \mathsf{ref}\ p \mid p := p' \mid\ !p \end{array}$$

The set of free variables of a program p is written $FV(p)$, and the set of locations occurring in p is written $\mathsf{loc}(p)$. We also use the shorthand $\mathsf{let}\ x = p\ \mathsf{in}\ p'$ (local definition) for $(\lambda x.\, p')\ p$, the same program being more simply written $p; p'$ (sequence) in the case where $x \notin FV(p')$. The fixpoint combinator for call-by-value strategy can also be encoded as $\mathsf{fix} \equiv \lambda f.\ (\lambda x.\ f\ (\lambda y.\ xxy))\ \lambda x.\ f\ (\lambda y.\ xxy)$.

We call a *value* (notation: v, v', v_1, etc.) any closed program that is generated from the following grammar:

$$v \quad ::= \quad n \mid \ell \mid (v_1,\, v_2) \mid \lambda x.\, p \qquad\qquad (FV(p) \subseteq \{x\})$$

The set of all values is written $\mathcal{V}$. This set is equipped with an equivalence relation, written $v \sim v'$, that is used to implement the structural equality test. The definition of this relation will be given in Section 5.

3.2 Operational Semantics

Stores We call a *store* any partial function $s : \mathcal{L} \rightharpoonup \mathcal{V}$ whose domain, written $\mathsf{dom}(s)$, is finite. A store may either represent the contents of the memory, or simply a set of local modifications (a 'patch'). In this spirit, we define an operation of *asymmetric merge* between stores, written $s_1 \Subset s_2$ and defined by

$$\begin{array}{rcll} \mathsf{dom}(s_1 \Subset s_2) & = & \mathsf{dom}(s_1) \cup \mathsf{dom}(s_2) & \\ (s_1 \Subset s_2)(\ell) & = & s_2(\ell) & \text{if } \ell \in \mathsf{dom}(s_2) \\ (s_1 \Subset s_2)(\ell) & = & s_1(\ell) & \text{otherwise} \end{array}$$

Intuitively, $s_1 \Subset s_2$ is the store obtained by applying the 'patch' s_2 to s_1. This operation will be used in the semantics of the update mechanism in Sect. 5.

In what follows, we assume given an *allocation function* alloc that associates to every store s a new location $\mathsf{alloc}(s) \in \mathcal{L}$ such that $\mathsf{alloc}(s) \notin \mathsf{dom}(s)$.

Evaluation contexts Evaluation contexts specify the strategy of evaluation. They are defined from the BNF:

$$\begin{array}{rcl} \mathsf{C} & ::= & () \mid \mathsf{f}(v_1, \ldots, v_n, \mathsf{C}, p_1, \ldots, p_m) \mid \mathsf{C} = p \mid v = \mathsf{C} \\ & \mid & (\mathsf{C},\, p) \mid (v,\, \mathsf{C}) \mid \mathsf{fst}(\mathsf{C}) \mid \mathsf{snd}(\mathsf{C}) \mid \mathsf{C}\, p \mid v\, \mathsf{C} \\ & \mid & \mathsf{if}\ \mathsf{C}\ \mathsf{then}\ p_1\ \mathsf{else}\ p_2 \mid \mathsf{ref}\ \mathsf{C} \mid\ !\mathsf{C} \mid \mathsf{C} := p \mid v := \mathsf{C} \end{array}$$

We assume programs p, p_1, p_2 occurring in the above definition are closed, so evaluation contexts are closed objects. Similarly, we assume that the function symbols f are totally applied.[5] We write $\mathsf{C}(p)$ for the (closed) program obtained by substituting the (closed) program p to the hole $()$ in the evaluation context C.

[5] According to this definition, arguments of functions are thus evaluated from the left to the right, as well as members of equalities, components of pairs, etc.

Evaluation An *evaluation state* is a pair $p \star s$ formed by a closed program p and a store s. The relation of one-step evaluation, written $p \star s \succ p' \star s'$, is the binary relation over evaluation states that is defined from the axioms of Figure 1, plus the 'context' rule

$$\frac{p \star s \quad \succ \quad p' \star s'}{\mathsf{C}(p) \star s \quad \succ \quad \mathsf{C}(p') \star s'}$$

We denote with $\succ^*$ the reflexive-transitive closure of the relation $\succ$.

$$(\lambda x.\ p)\ v \star s \ \succ\ {}^{[v}/_{x]}\ p \star s \qquad f(n_1, \ldots, n_k) \star s \ \succ\ \tilde{f}(n_1, \ldots, n_k) \star s$$
$$\mathsf{fst}(v_1, v_2) \star s \ \succ\ v_1 \star s \qquad \mathsf{if}\ n\ \mathsf{then}\ p_1\ \mathsf{else}\ p_2 \star s \ \succ\ p_1 \star s \quad (\text{if } n \neq 0)$$
$$\mathsf{snd}(v_1, v_2) \star s \ \succ\ v_2 \star s \qquad \mathsf{if}\ 0\ \mathsf{then}\ p_1\ \mathsf{else}\ p_2 \star s \ \succ\ p_2 \star s$$

$$v = v' \star s \ \succ\ \begin{cases} 1 \star s & \text{if } v \sim v' \\ 0 \star s & \text{if } v \not\sim v' \end{cases} \qquad \begin{array}{rcll} \mathsf{ref}\ v \star s & \succ & \ell \star (s \Subset \ell \leftarrow v) & (\text{if } \ell = \mathsf{alloc}(s)) \\ \ell := v \star s & \succ & 0 \star s \Subset \ell \leftarrow v & (\text{if } \ell \in \mathsf{dom}(s)) \\ !\ell \star s & \succ & s(\ell) \star s & (\text{if } \ell \in \mathsf{dom}(s)) \end{array}$$

$$\begin{array}{rcll} v_1\, v_2 \star s & \succ & 0 \star s & (\text{if } v_1 \text{ is not an abstraction}) \\ f(v_1, \ldots, v_k) \star s & \succ & 0 \star s & (\text{if } v_i \notin \mathbb{Z} \text{ for some } i \in [1..k]) \\ \mathsf{if}\ v\ \mathsf{then}\ p_1\ \mathsf{else}\ p_2 \star s & \succ & 0 \star s & (\text{if } v \notin \mathbb{Z}) \end{array}$$

$$\begin{array}{rcll} \mathsf{fst}(v) \star s & \succ & 0 \star s & (\text{if } v \text{ is not a pair}) \\ \mathsf{snd}(v) \star s & \succ & 0 \star s & (\text{if } v \text{ is not a pair}) \end{array} \qquad \begin{array}{rcll} v := v' \star s & \succ & 0 \star s & (\text{if } v \notin \mathsf{dom}(s)) \\ !v \star s & \succ & 0 \star s & (\text{if } v \notin \mathsf{dom}(s)) \end{array}$$

Fig. 1. One step evaluation rules

Note that the evaluation rules given above (that are clearly deterministic) explicitly deal with 'runtime errors' (such as applying a value that is not a function, etc.) and return the arbitrary value 0 in this case. This leads to the following lemma which guarantees correctness of logical rules (in particular BOX-NCS rule of section 4.5).

Lemma 1 (Determinism and progression). *For all evaluation states $p \star s$, there is at most one evaluation state $p' \star s'$ such that $p \star s \succ p' \star s'$. Moreover, this evaluation state $p' \star s'$ exists if and only if p is a not a value.*

3.3 Well-formedness of stores

Let s be a store. A program (or a value) p is *well-formed* in the store s when $\mathsf{loc}(p) \subseteq \mathsf{dom}(s)$. The set of well-formed values in s is written $\mathcal{V}_s$. A *well-formed store* is a store s such that $s(\ell) \in \mathcal{V}_s$ for all $\ell \in \mathsf{dom}(s)$. The set of well-formed stores is written $\mathbb{S}$. Finally, an evaluation state $p \star s$ is said to be *well-formed* when s is a well-formed store and p is well-formed in s. Well-formedness of evaluation states is preserved by evaluation:

Lemma 2. *If $p \star s$ is a well-formed evaluation state and $p \star s \succ p' \star s'$, then $p' \star s'$ is a well-formed evaluation state too.*

4 Logical System

We present the syntax and the rules of a proof language designed to specify programs such as defined in Sect. 3. This proof language is based on an extension

of Dynamic Logic (DL) with second-order quantifications, so that the language includes second-order functional arithmetic (AF2) [9] as well as the modalities of DL. The individuals manipulated by this logic are *symbolic values* that are formally defined below. Programs (actually: symbolic programs) may also appear inside formulas but restricted to specific positions as we shall see.

4.1 Symbolic Expressions

Location Names To reason efficiently about locations without mentioning them explicitly in the specification language, we introduce a new category of names, called *location names* and written α, β, γ, etc. Semantically, location names are characterized by three invariants:

1. A location name always refers to a concrete location.
2. The location referred to by a name is always allocated in the current store.
3. Two distinct location names refer to two distinct locations.

These invariants are essential to deal with problems of freshness and aliasing, and to ensure the absence of memory faults during evaluation (see Sect. 5).

Symbolic Programs Symbolic programs are defined in the same way as the programs introduced in Sect. 3. The only difference is that concrete locations are replaced by location names in the BNF. In this section p, q, p', etc., denote symbolic programs instead of concrete programs.

The (capture-preserving) implicit substitution operation is defined as in the λ-calculus, and its result is written $[{}^{p'}/_{x}]p$. Note that in presence of side effects, this operation is not semantically sound, since the programs $[{}^{p'}/_{x}]p$ and $\mathsf{let}\ x = p'\ \mathsf{in}\ p$ do not generally have the same operational semantics. A counter-example is given by the program $[{}^{!r}/_{y}](\lambda x.\ y) \equiv \lambda x.\ !r$, that does not behave the same way as the program $\mathsf{let}\ y = !r\ \mathsf{in}\ \lambda x.\ y$. For this reason, we shall put severe restrictions on the use of this form of substitution in the logic.

Symbolic Values Symbolic values form a sub-class of the syntactic category of symbolic programs, that is defined from the following BNF:

$$\begin{array}{rcl} \mathsf{v} & ::= & x \mid \alpha \mid n \mid \mathsf{f}(\mathsf{v}_1, \ldots, \mathsf{v}_n) \mid \mathsf{v}_1 = \mathsf{v}_2 \mid \lambda x.\ p \\ & \mid & (\mathsf{v}_1,\ \mathsf{v}_2) \mid \mathsf{fst}(\mathsf{v}) \mid \mathsf{snd}(\mathsf{v}) \mid \mathsf{if}\ \mathsf{v}\ \mathsf{then}\ \mathsf{v}_1\ \mathsf{else}\ \mathsf{v}_2 \end{array}$$

(Unlike concrete ML-values, symbolic values may be open as well as closed.)

Intuitively, symbolic values correspond to the programs that do not access the store, and whose form is simple enough to ensure termination. For this reason, every symbolic value unambiguously refers to a concrete value (provided we assign a value to every variable and a location to every location name).

Substitution of symbolic values v is thus a safe operation, since the program $[{}^{\mathsf{v}}/_{x}]\ p$ has the same semantics as $\mathsf{let}\ x = \mathsf{v}\ \mathsf{in}\ p$.

$$
\begin{array}{rcl}
\mathsf{f}(n_1,\ldots,n_k) & \cong & \tilde{\mathsf{f}}(n_1,\ldots,n_k) \\
(\lambda x.\ p)\ \mathsf{v} & \cong & {}^{[\mathsf{v}}/_{x]}\ p \\
\mathsf{fst}((\mathsf{v}_1,\ \mathsf{v}_2)) & \cong & \mathsf{v}_1 \\
\mathsf{snd}((\mathsf{v}_1,\ \mathsf{v}_2)) & \cong & \mathsf{v}_2
\end{array}
\qquad
\begin{array}{rcll}
\mathsf{if}\ n\ \mathsf{then}\ p\ \mathsf{else}\ p' & \cong & p & (n \neq 0) \\
\mathsf{if}\ 0\ \mathsf{then}\ p\ \mathsf{else}\ p' & \cong & p' & \\
\mathsf{v} = \mathsf{v} & \cong & 1 & \\
n = m & \cong & 0 & (n \neq m) \\
\alpha = \beta & \cong & 0 & (\alpha \neq \beta)
\end{array}
$$

Fig. 2. Congruence on symbolic programs

Symbolic Evaluation of Symbolic Programs The class of symbolic programs comes with a congruence written $p \cong p'$ that expresses that the two programs p and p' are equivalent modulo zero, one or several steps of symbolic evaluation. This congruence is defined from rules of Fig 2:

Note that these rules can be applied in any context, even under λ-abstractions. In particular, we have $\lambda x.\ 1+1 \ \cong\ \lambda x.\ 2$ even though both members are values that are not further evaluated.[6] The main reason for this design choice is that it makes the definition of the logical system conceptually and technically much more simple. (However, we shall see in Sect. 5.1 that this choice has subtle consequences on the semantics.)

4.2 Updates

We employ a simplified form of update as compared to the general definition in [11]. Formally, updates (notation: u, u', u_1, etc.) are defined as finite lists of pairs of symbolic values of the form $\mathsf{v} := \mathsf{v}'$:

$$u \quad ::= \quad \emptyset \quad | \quad u\,;\,\mathsf{v} := \mathsf{v}'$$

(Note that $\emptyset$ acts a neutral element, hence $\emptyset\,;\,u \ \equiv\ u$.) The application of an update u to a symbolic program of the form $!\mathsf{v}$ (where v is a symbolic value) is written $\{u\}!\mathsf{v}$ and defined by

$$
\begin{array}{rcl}
\{\emptyset\}!\mathsf{v} & = & !\mathsf{v} \\
\{u\,;\,\mathsf{v}_1 := \mathsf{v}_2\}!\mathsf{v} & = & \mathsf{if}\ \mathsf{v} = \mathsf{v}_1\ \mathsf{then}\ \mathsf{v}_2\ \mathsf{else}\ \{u\}!\mathsf{v}
\end{array}
$$

Note that the result of this operation is a symbolic program that can be simplified using the congruence rules of symbolic evaluation.

4.3 Formulas

Formulas (notation: A, B, C, etc.) are built from second-order variables (notation: X, Y, Z, etc.) that represent k-ary relations. We assume that every second-order variable comes with an arity which we indicate as a superscript when we introduce the variable. The syntax of formulas is the following:

$$
\begin{array}{rcl}
A & ::= & X(\mathsf{v}_1,\ldots,\mathsf{v}_k) \quad | \quad A \to B \quad | \quad \forall x.\ A \quad | \quad \forall X^k.\ A \\
& | & I(\mathsf{v}) \quad | \quad \nu\alpha.\ A \quad | \quad [p\ \ \mathsf{as}\ x]\,A \quad | \quad \{u\}A
\end{array}
$$

[6] The integer addition is included in the set of primitive functions as well as all standard arithmetic operations (see section 3.1).

(For simplicity, we consider a language based on implication and first- and second-order universal quantification, from which we easily recover other connectives and quantifiers.) We also provide the following constructs:

- A predicate constant I that transforms any symbolic value v into a formula $I(\mathsf{v})$ that is true if the concrete value denoted by v is a value different from 0.
- A construct $[p \ \mathsf{as}\ x]\,A$ that means: 'if p evaluates to a value x, then A holds in the store affected by all the side effects performed by p'. This construction is nothing but the box modality of DL that we transformed into a binder to recover the value computed by the program p. In particular, when A does not depend on x, we simply write $[p]A$.
- A construct $\{u\}A$ that means: 'after updating the current store with the assignments in u, A holds'.
- A construct $\nu\alpha.A$ (ν-binder) that means: 'after the allocation of a fresh address named α, A holds'.

The set of free variables (free names) of a formula A is written $\mathit{FV}(A)$ ($\mathit{FN}(A)$).

4.4 Symbolic Evaluation

The congruence defined in Sect. 4.1 over symbolic programs is extended to formulas which, together with a contextual closure, occur within formulas and with specific rules for decomposing boxes as well as for propagating updates and νs throughout the structure of formulas (Fig. 3).

$$
\begin{array}{rcll}
I(0) & \cong & \bot\ (\equiv \forall X.X) & \\
I(n) & \cong & \top\ (\equiv \forall X.X \to X) & n \neq 0 \\
\multicolumn{4}{c}{\textbf{Decomposition of boxes}} \\
[\mathsf{C_{se}}(p) \ \mathsf{as}\ x]\,A & \cong & [p \ \mathsf{as}\ y]\,[\mathsf{C_{se}}(y) \ \mathsf{as}\ x]\,A & y \notin \mathit{FV}(\mathsf{C_{se}}(p), A, x) \\
[\mathsf{ref}\ \mathsf{v} \ \mathsf{as}\ x]\,A & \cong & \nu\alpha.\{\alpha := \mathsf{v}\}^{[\alpha/x]}\ A & \alpha \notin \mathit{FN}(A, \mathsf{v}) \\
[\mathsf{v}_1 := \mathsf{v}_2]A & \cong & \{\mathsf{v}_1 := \mathsf{v}_2\}A & \\
[\mathsf{v} \ \mathsf{as}\ x]\,A & \cong & {}^{[\mathsf{v}/x]}\ A & \\
\multicolumn{4}{c}{\textbf{Propagation of updates}} \\
\{u\}I(\mathsf{v}) & \cong & I(\mathsf{v}) & \\
\{u\}(A \to B) & \cong & \{u\}A \to \{u\}B & \\
\{u\}\forall x.A & \cong & \forall x.\{u\}A & x \notin \mathit{FV}(u) \\
\{u\}\forall X.A & \cong & \forall X.\{u\}A & \\
\{u\}\nu\alpha.A & \cong & \nu\alpha.\{u\}A & \alpha \notin \mathit{FN}(u) \\
\{u\}\{u'\}A & \cong & \{u\,;\,u'\}A & \\
\{u\}[!\mathsf{v} \ \mathsf{as}\ x]\,A & \cong & [\{u\}!\mathsf{v} \ \mathsf{as}\ x]\,\{u\}A & x \notin \mathit{FV}(u) \\
\multicolumn{4}{c}{\textbf{Propagation of } \nu\textbf{s}} \\
\forall X^n.\ \nu\alpha.A & \cong & \nu\alpha.\forall X^n.\ A & \\
[p \ \mathsf{as}\ x]\,\nu\alpha.A & \cong & \nu\alpha.[p \ \mathsf{as}\ x]\,A & \alpha \notin \mathit{FN}(p) \\
\nu\alpha.\nu\beta.A & \cong & \nu\beta.\nu\alpha.A &
\end{array}
$$

Fig. 3. Symbolic evaluation of formulas

Decomposition of boxes The decomposition of boxes has to take care of the evaluation order. The first rule splits a program inside a box in two pieces according to a given symbolic evaluation context C_{se}. (Symbolic evaluation contexts are defined as for evaluation contexts, replacing explicit locations with location names and explicit values with symbolic values.) Note that the enclosing symbolic evaluation context is not uniquely determined by the program within the box, and this rule can be used to decompose the very same box in many different ways. The next two rules deal with the creation of a reference (that introduces a ν-binder and an update) and with an assignment (that introduces an update). The last rule simply removes a box when the inner program is a symbolic value.

Propagation of updates Updates go down through the structure of formulas until they reach a box. An update can go through a box only when the inner program is of the form $!\mathsf{v}$ (access to the contents of a reference), in which case the program is updated using the construction $\{u\}!\mathsf{v}$ defined in Section 4.2. In all the other cases, the update is stuck in front of the box until this box is decomposed into smaller boxes using symbolic evaluation.

Propagation of νs The ν-binder comes with quite standard propagation rules (we do not give them all). Note that there is a rule for commuting a ν-binder with second-order quantification, but no analogous rule for first-order quantification. The reason is that semantically, the domain of first-order quantification depends on the set of currently allocated locations, so that we have

$$\forall x.\nu\alpha.A \quad \not\to \quad \nu\alpha.\forall x.A$$

in general. We shall come back to this point in Sect. 5. Note also that in general a ν-binder cannot be dropped even when the name it binds does not occur in its scope, so we have $\nu\alpha.A \not\to A$ even if $\alpha \notin \mathit{FN}(A)$.

4.5 Deduction Rules

The language is equipped with a Gentzen-style sequent calculus. This system includes the standard rules for second-order logic: structural rules (weakening and contraction), axiom, cut, plus the standard left and right rules for implication, first- and second-order universal quantification.

$$\frac{\Gamma, A' \vdash \Delta \qquad A \cong A'}{\Gamma, A \vdash \Delta}\ \text{RWG} \qquad \frac{\Gamma \vdash A', \Delta \qquad A \cong A'}{\Gamma \vdash A, \Delta}\ \text{RWD}$$

$$\frac{\Gamma \vdash \Delta}{\nu\alpha.\ \Gamma \vdash \nu\alpha.\ \Delta}\ \nu\text{NCS} \qquad \frac{\Gamma \vdash \Delta}{\{u\}\Gamma \vdash \{u\}\Delta}\ \text{UPD-NCS}$$

$$\Delta \neq \emptyset\ \frac{\Gamma \vdash \Delta}{[p \ \mathsf{as}\ x]\,\Gamma \vdash [p \ \mathsf{as}\ x]\,\Delta}\ \text{BOX-NCS}$$

Fig. 4. Specific deduction rules

The specific rules of our system (see Fig. 4) include:
- Left and right rules for symbolic evaluation, expressing that computationally equivalent formulas (via symbolic evaluation) are logically equivalent. (Because of looping programs, the verification of an instance of these rules is not decidable.[7])
- Necessitation rules for all modalities (ν-binder, updates, and boxes).

Note that the generalized forms of the standard necessitation rules are allowed in our case because the programing language is deterministic and because values are normal forms (Lemma 1), so that the frame relation underlying each modality (including updates) is functional. The side condition of BOX-NCS is necessary because the evaluation of the inner program might not terminate. In this case, the hypothesis $[p \ \mathsf{as}\ x]\,\Gamma$ becomes vacuously valid (as we shall see in Sect. 5) while the empty conclusion is obviously not.

5 Semantics

We now build a Kripke model of the language where worlds are well-formed stores (simply called stores from now on). In this setting, each symbolic value is interpreted as a concrete value whereas each formula is interpreted as the set of stores in which the formula is true. The construction is standard, with some subtleties that will be explained in Sect. 5.1. The main feature of the model is that the domain of interpretation of the individuals (i.e. symbolic values) has to depend on the current store, because the values which make sense in a store s are those which are well-formed in s. (In particular, this property explains why we cannot commute first-order quantification with ν-binders.)

5.1 Invariance properties

Equivalence of values Both in IML and in the logical framework, two functional values $\lambda x.\ p$ and $\lambda x.\ p'$ are observationally equivalent when $p \cong p'$. To identify such values in the model, we introduce the relation of *equivalence of values*, written $v \sim v'$, as the least equivalence relation such that:
- If $p \cong p'$, then $\lambda x.\ p \sim \lambda x.\ p'$.
- If $v_1 \sim v_1'$ and $v_2 \sim v_2'$, then $(v_1, v_2) \sim (v_1', v_2')$.

Invariance under automorphisms Similarly, the allocation order of locations is indistinguishable both for IML programs and for the logical framework. In order to ensure that the model is not sensitive to the allocation order either, we need to introduce the notion of invariance under all automorphisms.

Formally, an *automorphism* (of locations) is any bijection σ over the set $\mathcal{L}$ of locations. An automorphism σ can be applied to a location, but also to a value (by applying σ to all the locations inside the value) as well as to a store.

Formally, the store $\sigma(s)$ is defined by $\mathsf{dom}(\sigma(s)) = \sigma(\mathsf{dom}(s))$ and

$$\sigma(s)(\ell) \ = \ \sigma(s(\sigma^{-1}(\ell))) \qquad\qquad (\ell \in \mathsf{dom}(\sigma(s)))$$

[7] In an implementation, it will be possible to force arbitrarily the decision, for example by limiting the number of evaluation steps.

Propositional functions Let $f : \mathcal{V}^k \to \mathfrak{P}(\mathbb{S})$ be a function from k-tuples of values to sets of (well-formed) stores. We say that f is:

- *compatible with the equivalence of values* when for all $v_1, \ldots, v_k, v'_1, \ldots, v'_k$ such that $v_1 \sim v'_1, \cdots, v_k \sim v'_k$: $f(v_1, \ldots, v_k) = f(v'_1, \ldots, v'_k)$.
- *invariant under all automorphisms* when for all $v_1, \ldots v_k \in \mathcal{V}$, $s \in \mathbb{S}$ and for all automorphisms σ: $s \in f(v_1, \ldots, v_k)$ iff $\sigma(s) \in f(\sigma(v_1), \ldots, \sigma(v_k))$.

The set of all functions $f : \mathcal{V}^k \to \mathfrak{P}(\mathbb{S})$ that are both compatible with the equivalence of values and invariant under all automorphisms is written $\mathcal{F}^k_{\mathcal{P}}$. In what follows, we shall interpret predicates variables of arity k (and formulas depending on k first-order variables) as elements of $\mathcal{F}^k_{\mathcal{P}}$.

5.2 Interpreting symbolic values and updates

Valuations A *valuation* is a function ρ that maps each

- first-order variable x to a value $\rho(x) \in \mathcal{V}$;
- k-ary second-order variable X to a propositional function $\rho(x) \in \mathcal{F}^k_{\mathcal{P}}$;
- location name α to a location $\rho(\alpha) \in \mathcal{L}$.

Moreover, we require that ρ is injective on location names: distinct location names are mapped to distinct locations. A valuation ρ is *well-formed* in a store s when $\rho(x) \in \mathcal{V}_s$ for all $x \in \mathsf{dom}(\rho)$ and $\rho(\alpha) \in \mathsf{dom}(s)$ for all $\alpha \in \mathsf{dom}(\rho)$. This notion is clearly preserved by store extension.

Interpreting symbolic values Given a symbolic value v and a valuation ρ, we denote by $[\![\mathsf{v}]\!]_\rho$ the unique value v such that $\mathsf{v}[\rho] \star s \succ^* v \star s$, where s is an arbitrary store. Note that such a value always exists—due to the restricted form of symbolic values—and that it is unique since evaluation is deterministic. Moreover, the value v does not depend on s, and the evaluation of the program $\mathsf{v}[\rho]$ that computes the value v does not modify the store.

Interpreting updates Updates are interpreted as stores (intuitively: as 'patches' to the global memory). Given an update u and a valuation ρ, we define $[\![u]\!]_\rho$ by:

$$[\![\emptyset]\!]_\rho \;=\; \emptyset \qquad \text{and} \qquad [\![u\,;\mathsf{v}_1 := \mathsf{v}_2]\!]_\rho \;=\; [\![u]\!]_\rho \Subset [\![\mathsf{v}_1]\!]_\rho \leftarrow [\![\mathsf{v}_2]\!]_\rho$$

5.3 Interpreting formulas and sequents

The relation of satisfiability of formulas (where s is a well-formed store and ρ a valuation that is well-formed in s) is defined in Fig. 5:

The interpretation immediately extends to sequents (notation $s \models (\Gamma \vdash \Delta)[\rho]$) reading left hand-side commas as conjunctions, right hand-side commas as disjunctions and the symbol '$\vdash$' as implication. Note that the formula $[p \;\; \mathsf{as}\; x]\, A$ is always valid when p does not terminate.

Theorem 1 (Correctness of the system) *If the sequent $\Gamma \vdash \Delta$ is derivable in the system, then for all well-formed stores $s \in \mathbb{S}$ and for all valuations ρ that are well-formed in s, we have $s \models (\Gamma \vdash \Delta)[\rho]$.*

$$
\begin{array}{lll}
s \models X(\mathsf{v}_1,\ldots,\mathsf{v}_k)[\rho] & \text{iff} & s \in \rho(X)(\llbracket \mathsf{v}_1 \rrbracket_\rho,\ldots,\llbracket \mathsf{v}_k \rrbracket_\rho) \\
s \models I(\mathsf{v})[\rho] & \text{iff} & \llbracket \mathsf{v} \rrbracket_\rho \neq 0 \\
s \models (A \to B)[\rho] & \text{iff} & s \models A[\rho] \ \text{ implies } \ s \models B[\rho] \\
s \models (\forall x.A)[\rho] & \text{iff} & \text{for all } v \in \mathcal{V}_s, \ \ s \models A[\rho\,;\, x \mapsto v] \\
s \models (\forall X^k.\ A)[\rho] & \text{iff} & \text{for all } f \in \mathcal{F}^k_{\mathcal{P}}, \ s \models A[\rho\,;\, X \mapsto f] \\
s \models (\nu\alpha.A)[\rho] & \text{iff} & (s \otimes \mathsf{alloc}(s) \leftarrow 0) \models A[\rho\,;\, \alpha \mapsto \mathsf{alloc}(s)] \\
s \models (\{u\}A)[\rho] & \text{iff} & s \otimes \llbracket u \rrbracket_\rho \models A[\rho] \\
s \models ([p \ \ \mathsf{as}\ x]\,A)[\rho] & \text{iff} & \text{for all } s' \in \$ \text{ and } v \in \mathcal{V}_{s'}, \\
 & & p[\rho] \star s \succ^* v \star s' \ \text{ implies } \ s' \models A[\rho\,;\, x \mapsto v]
\end{array}
$$

Fig. 5. Satisfiability of formulas

Theorem 1 relies on many intermediate lemmas that are not given here. Basically, these lemmas express that all the constructions of the programming language and of the logical framework are compatible with the equivalence of values and preserve the property of invariance under all automorphisms.

It is easy to check that $s \not\models \bot$. Hence, the formula $\bot$ cannot be proved within our system, which is thus consistent.

6 Specification and Verification of a Recursive Function

We illustrate how to specify and verify a recursive function along a small example. Let us now consider the program cc defined by

$$
\begin{array}{lll}
cc & \equiv & \lambda n.\ \mathsf{let}\ c = (\mathsf{let}\ r = \mathsf{ref}\ 0\ \mathsf{in}\ \lambda x.\ \ r :=\!!r + 1\ ;!r)\ \mathsf{in} \\
 & & \quad \mathsf{let}\ aux = \mathsf{fix}\ (\lambda f n.\ \mathsf{if}\ n = 0\ \mathsf{then}\ c\ 0\ \mathsf{else}\ (c\ 0\ ;\ f\ (n-1)))\ \mathsf{in} \\
 & & \quad aux\ n
\end{array}
$$

This program takes a natural number n as an argument and calls $n+1$ times the sub-program c that contains a local reference, before returning the result of the last call of c. (Here the argument 0 in $c\ 0$ plays the role of `()` in ML.)

We intend to prove that for all natural numbers n, the result of $cc\ n$ is $n+1$. Therefore, we need first to characterise natural numbers among all the possible values. For the characterisation we use their well-known second-order definition as given in [9]: $\mathsf{Nat}(x) \ \equiv\ \forall X.\ (\forall y.(X(y) \to X(y+1)) \to X(0) \to X(x))$

For readability, we introduce the notation $\forall x : \mathsf{Nat}.\ A$ (relativized quantification) for $\forall x.(\mathsf{Nat}(x) \to A)$. The property of interest can be expressed by:

$$\forall n : \mathsf{Nat}.\ [\mathrm{cc}\ n = n+1 \ \ \mathsf{as}\ b]\, I(b)\,.$$

A derivation (Π_1) is shown in Fig. 7. We use the obvious rules that can be derived from the definition of $\mathsf{Nat}(x)$ as well as the (derived) induction rule:

$$\frac{\vdash\ A(0) \qquad \mathsf{Nat}(n),\ A(n)\ \vdash\ A(n+1)}{\vdash\ \forall n : \mathsf{Nat}.\ A(n)}\ \mathrm{Ind}$$

To simplify Π_1, we denote by $\underline{\mathsf{aux}}$ and $\underline{\mathsf{aux}'}$ (of Fig. 6) the functional values these programs reduce to. The specification is proved using an auxiliary lemma (L) stating that the property holds for any content of the reference: this lemma is proved (left premise of the Cut rule in Π_1) by induction. Note that this lemma can be used only in a context in which the inner reference is visible. The bottom

$$
\begin{array}{lcl}
\underline{\mathsf{c}} & \equiv & \mathsf{let}\ r = \mathsf{ref}\ 0\ \mathsf{in}\ \lambda x.\ r := !r + 1\ ;!r \\
\underline{\mathsf{aux}} & \equiv & \mathsf{fix}\ (\lambda f n.\ \mathsf{if}\ n = 0\ \mathsf{then}\ c\ 0\ \mathsf{else}\ (c\ 0\ ;f\ (n-1))) \\
\underline{\mathsf{cc}} & \equiv & \lambda n.\ \mathsf{let}\ c = \underline{\mathsf{c}}\ \mathsf{in}\ \mathsf{let}\ aux = \underline{\mathsf{aux}}\ \mathsf{in}\ aux\ n \\
\underline{\mathsf{aux}'} & \equiv & \mathsf{fix}\ (\lambda f n.\ \mathsf{if}\ n = 0\ \mathsf{then}\ (\lambda x.\ \alpha := !\alpha + 1\ ;!\alpha)\ 0 \\
 & & \qquad\qquad\qquad\qquad \mathsf{else}\ \ (\lambda x.\ \alpha := !\alpha + 1\ ;!\alpha)\ 0\ ;f\ (n-1))
\end{array}
$$

Fig. 6. Shortcuts

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{}{\vdash\ I(1)}\ \mathrm{Ax}
}{\vdash\ \{\alpha\ :=\ k+1\}I(1)}\ \cong
}{\vdash\ \{\alpha\ :=\ k+1\}[k+1 = k+1\ \ \mathsf{as}\ b]\, I(b)}\ \cong \times 2
}{\vdash\ \{\alpha\ :=\ k+1\}[!\alpha = k+1\ \ \mathsf{as}\ b]\, I(b)}\ \cong
}{\vdash\ \{\alpha\ :=\ k\}[(\alpha := k+1;!\alpha) = k+1\ \ \mathsf{as}\ b]\, I(b)}\ \cong
}{\vdash\ \{\alpha\ :=\ k\}[(\alpha := !\alpha+1;!\alpha) = k+1\ \ \mathsf{as}\ b]\, I(b)}\ \cong
}{\vdash\ \{\alpha\ :=\ k\}[\underline{\mathsf{aux}'}\ 0 = k+1\ \ \mathsf{as}\ b]\, I(b)}\ \cong
}{\vdash\ \forall k.\ \{\alpha\ :=\ k\}[\underline{\mathsf{aux}'}\ 0 = k+1\ \ \mathsf{as}\ b]\, I(b)}\ \forall\mathrm{R}
\qquad
\dfrac{[\Pi_3]}{RH\ \vdash\ RS}\ \mathrm{Ind}
\qquad
\dfrac{[\Pi_2]}{L,\ \mathsf{Nat}(n)\ \vdash\ C}
}{\vdash\ \forall n : \mathsf{Nat}.\ \forall k.\ \{\alpha\ :=\ k\}[\underline{\mathsf{aux}'}\ n = n+k+1\ \ \mathsf{as}\ b]\, I(b)}\ \mathrm{Cut}
}{\mathsf{Nat}(n)\ \vdash\ \{\alpha\ :=\ 0\}[\underline{\mathsf{aux}'}\ n = n+1\ \ \mathsf{as}\ b]\, I(b)}\ \forall\mathrm{R}
}{\mathsf{Nat}(n)\ \vdash\ \nu\alpha.\{\alpha\ :=\ 0\}[\underline{\mathsf{aux}'}\ n = n+1\ \ \mathsf{as}\ b]\, I(b)}\ \nu\mathrm{R}
}{\mathsf{Nat}(n)\ \vdash\ \nu\alpha.\{\alpha\ :=\ 0\}[\lambda x.\ \alpha := !\alpha+1;!\alpha\ \ \mathsf{as}\ c]\,[\underline{\mathsf{aux}}\ n = n+1\ \ \mathsf{as}\ b]\, I(b)}\ \cong
}{\mathsf{Nat}(n)\ \vdash\ [\mathsf{ref}\ 0\ \ \mathsf{as}\ r]\,[\lambda x.\ r := !r+1;!r\ \ \mathsf{as}\ c]\,[\underline{\mathsf{aux}}\ n = n+1\ \ \mathsf{as}\ b]\, I(b)}\ \cong
}{\mathsf{Nat}(n)\ \vdash\ [\underline{\mathsf{c}}\ \ \mathsf{as}\ c]\,[\underline{\mathsf{aux}}\ \ \mathsf{as}\ aux]\,[aux\ n = n+1\ \ \mathsf{as}\ b]\, I(b)}\ \mathrm{Let} + \cong
}{\mathsf{Nat}(n)\ \vdash\ [\mathsf{let}\ c = \underline{\mathsf{c}}\ \mathsf{in}\ \mathsf{let}\ aux = \underline{\mathsf{aux}}\ \mathsf{in}\ aux\ n = n+1\ \ \mathsf{as}\ b]\, I(b)}\ \mathrm{Let} \times 2 + \cong
}{\mathsf{Nat}(n)\ \vdash\ [\underline{\mathsf{cc}}\ n = n+1\ \ \mathsf{as}\ b]\, I(b)}\ \cong
}{\vdash\ \forall n : \mathsf{Nat}.\ [\underline{\mathsf{cc}}\ n = n+1\ \ \mathsf{as}\ b]\, I(b)}\ \forall\mathrm{R}
$$

where $RS \equiv \forall k.\{\alpha := k\}[\underline{\mathsf{aux}'}(n+1) = n+k+2\ \ \mathsf{as}\ b]\, I(b)$
$RH \equiv \mathsf{Nat}(n), \forall k.\{\alpha := k\}[\underline{\mathsf{aux}'}\ n = n+k+1\ \ \mathsf{as}\ b]\, I(b)$
$L \equiv \forall n : \mathsf{Nat}.\ \forall k.\ \{\alpha := k\}[\underline{\mathsf{aux}'}\ n = n+k+1\ \ \mathsf{as}\ b]\, I(b)$
$C \equiv \{\alpha := 0\}[\underline{\mathsf{aux}'}\ n = n+1\ \ \mathsf{as}\ b]\, I(b)$

Fig. 7. Π_1: Derivation of the specification

part of the proof partially evaluates the program $cc\ n$ to make the inner reference visible at the top level.

The proof of the recursive step (Π_3) is done using only $\cong, \forall\mathrm{R}, \forall\mathrm{L}$. The sequent $L,\ \mathsf{Nat}(n)\ \vdash\ C$ is trivially proved (Π_2) with instances of the $\forall$L rule.

7 Conclusion

We have presented a system for specifying and verifying imperative ML programs, whose specification language combines dynamic logic with second-order logic à la AF2. This combination illustrates the flexibility of DL, that can be adapted to many programming languages (here: imperative ML) and to many logical frameworks (here: second-order logic), thus making them benefit of the strength of symbolic evaluation and of its deep impact in proof automation.

The next step is to test our system by implementing it, for instance as a component of KeY or within another logical framework. Another natural research direction would be the integration of a static type system at the level of the logic, following the spirit of the system of strong typing in PAF!

Acknowledgements Many thanks are due to Yann Régis-Gianas for stimulating discussions and valuable advices.

References

1. M. Balser, W. Reif, G. Schellhorn, K. Stenzel, and A. Thums. Formal system development with KIV. In Tom Maibaum, editor, *Fundamental Approaches to Software Engineering*, volume 1783 of *LNCS*. Springer-Verlag, 2000.
2. S. Baro. Introduction to PAF!, a proof assistant for ml programs verification. In *TYPES*, pages 51–65, 2003.
3. B. Beckert, R. Hähnle, and P. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer, 2006.
4. A. Chlipala, G. Malecha, G. Morrisett, A. Shinnar, and R. Wisnesky. Effective interactive proofs for higher-order imperative programs. In *ICFP '09: Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, September 2009.
5. J.-C. Filliâtre. Why: a multi-language multi-prover verification tool. Research Report 1366, LRI, Université Paris Sud, March 2003.
6. D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. Foundations of Computing. MIT Press, October 2000.
7. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 583, October 1969.
8. J. Kanig and J.-C. Filliâtre. Who: A Verifier for Effectful Higher-order Programs. In *ACM SIGPLAN Workshop on ML*, Edinburgh, Scotland, UK, August 2009.
9. J. L. Krivine. *Lambda-calculus, types and models*. Masson, 1993.
10. Y. Régis-Gianas and F. Pottier. A Hoare logic for call-by-value functional programs. In Philippe Audebaud and Christine Paulin-Mohring, editors, *Mathematics of Program Construction, 9th Intl. Conf., MPC 2008, Marseille, France*, volume 5133 of *LNCS*, pages 305–335. Springer, 2008.
11. P. Rümmer. Sequential, parallel, and quantified updates of first-order structures. In *Logic for Programming, Artificial Intelligence and Reasoning*, volume 4246 of *LNCS*, pages 422–436. Springer, 2006.

Abstract compilation of object-oriented languages into coinductive CLP(X): when type inference meets verification*

Davide Ancona[1], Andrea Corradi[1], Giovanni Lagorio[1], and Ferruccio Damiani[2]

[1] DISI, University of Genova, Italy
{davide,lagorio}@disi.unige.it,andreac@unstable.it
[2] Dipartimento di Informatica, University of Torino, Italy
damiani@di.unito.it

Abstract. We propose a novel general approach for defining expressive type systems for object-oriented languages, based on abstract compilation of programs into coinductive constraint logic programs defined on a specific constraint domain X called *type domain*. In this way, type checking and type inference amount to resolving a certain goal w.r.t. the coinductive (that is, the greatest) Herbrand model of a logic program (that is, a Horn formula) with constraints over a fixed type domain X. In particular, we show an interesting instantiation where the constraint predicates of X are syntactic equality and subtyping over coinductive object and union types. The corresponding type system is so expressive to allow verification of simple properties like data structure invariants. Finally, we show a prototype implementation, written in Prolog, of the inference engine for coinductive CLP(X), which is parametric in the solver for the type domain X.

1 Introduction

Mapping type checking and type inference algorithms to logic programming is not a novel idea; however, less known are the advantages that such an approach can bring to the research areas of type systems and software verification.

Sulzmann and Stuckey [18] have shown that the generalized Hindley/Milner type inference problem HM(X) [14] can be mapped to inductive CLP(X): type inference of a program can be obtained by first translating it in a set of CLP(X) clauses, and then by resolving a certain goal w.r.t. such clauses. This result is not purely theoretical, indeed it has also some important practical advantages: maintaining a strict distinction between the translation phase and the logical inference one, when the goal and the constraints are solved, allows a much clearer specification of type inference and a more modular approach, since different type inference algorithms can be obtained by just modifying the translation phase, while reusing the same engine defined in the logical inference phase.

* This work has been partially supported by MIUR DISCO - Distribution, Interaction, Specification, Composition for Object Systems.

Abstract compilation [5, 1, 2] is a novel approach for specifying coinductive type systems by abstractly compiling the program to be analyzed into a Horn formula. Then, type checking and type inference amount to resolving a certain goal w.r.t. the coinductive Herbrand model of the formula generated by the abstract compilation.

In contrast to the approaches based on inductive LP and CLP, the coinductive approach allows the specification of much more expressive type systems and, therefore, seems more appropriate for defining type inference useful for proving simple properties like data structure invariants. The potential offered by coinductive CLP for software verification has been investigated very recently also by Saeedloei and Gupta [12], even though in the different context of verification of complex continuous real-time systems.

Abstract compilation is particularly interesting for type inference of object-oriented languages [1, 2] when union and object types are combined together. However, for implementing the logical inference phase, one has necessarily to restrict the type system to regular types and derivations (corresponding to infinite trees having a finite set of subtrees) and to explicitly introduce a subtyping relation [2, 3] capturing the notion of approximation.

In previous work [1, 4] we have formally defined and studied three different translations from core Java-like languages, similar to Featherweight Java (FJ) [10], to Horn formulas:

1. from a purely functional object-oriented language L,[3] to show that the approach can be used for defining standard typechecking of Java-like languages ([1], Figures 3 and 4);
2. from a version of L where nominal type annotations can be optionally omitted, to show how precise type inference can be obtained from abstract compilation, and how coinductive union and object types can be smoothly integrated with nominal types, by considering nominal type annotations as additional constraints imposed by the user ([1], Figures 5 and 6);
3. from an SSA [8] intermediate form of an imperative version of FJ, to show how abstract compilation can be deal with imperative features like variable and field assignment and iterative constructs, and how SSA φ functions can be naturally encoded with union types ([4], Figures 3, 4 and 5).

In this paper we propose a novel general approach for expressing type inference as abstract compilation of programs into coinductive CLP(X), where X is a specific constraint domain, called *type domain*, containing two constraint predicates: strong equivalence between types (coinciding to syntactic equality in most cases) and subtyping.

This paper has two main contributions. First, it presents a general schema for abstract compilation based on coinductive CLP(X), where X corresponds to a specific type domain (see Figure 1). The input is represented by the source program to be analyzed and by a query defined by the user in a high level language.

[3] A variant of FJ that has generalized explicit constructor declarations and primitive types, but no type casts.

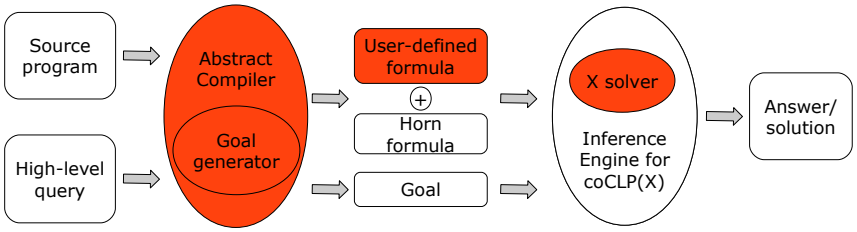


Fig. 1. General schema for abstract compilation based on coinductive CLP(X)

Then the abstract compiler and the goal generator, which is a subcomponent of the abstract compiler, generate a Horn formula (a conjunction of CLP clauses) and a goal. The generated clauses can be optionally augmented by user-defined clauses defining auxiliary predicates. Finally, type inference is performed by the coinductive CLP engine. The red (or dark) components are those depending on the type system under consideration: the abstract compiler[4] and the solver for the specific type domain X.

The second contribution is the definition of the operational semantics of coinductive CLP and of a corresponding implementation based on a Prolog prototype meta-interpreter, parametric in the solver for the type domain X. The implementation exploits variance annotations of user-defined predicate to use subsumption instead of simple term unification when the coinductive hypothesis rule is applied.

The paper is organized as follows. Section 2 provides some minimal background on coinductive LP and on inductive CLP. Section 3 explains abstract compilation, defines a type domain based on union and object types, and shows the expressive power of the system with some examples. Section 4 is devoted to the semantics and implementation of coinductive CLP, whereas Section 5 draws some conclusions and outlines some directions for further investigation.

2 Background

Coinductive LP and SLD Simon et al [17] have introduced *coinductive-LP*, or simply *co-LP*. Its declarative semantics is given in terms of *co-Herbrand* universe, infinitary Herbrand base and maximal models, computed using greatest fixed-points. While in traditional LP this semantics corresponds to build finite proof trees, co-LP allows infinite terms and proofs as well, which in general are not finitely representable and, for this reason, are called idealized. The operational semantics, defined in a manner similar to SLD, is called *co-SLD*. For an

[4] If the source language is unchanged only the back-end will be modified.

obvious reason, co-SLD is restricted to *regular* terms and proofs, that is, to trees which may be infinite, but can only contain a finite number of different subtrees (and, hence, can be finitely represented). To correctly deal with infinite regular derivations an implicit *coinductive hypothesis rule* is introduced. This rule allows a predicate call to succeed if it unifies with one of its ancestor calls.

CLP(X) CLP introduces *constraints* in the body of the clauses of a logic program, specifying conditions under which the clauses hold, and let external constraint solvers interpret/simplify these constraints. For instance, the clause $p(X) \leftarrow \{X > 3\},\ q(X)$ expresses that $p(X)$ holds when $q(X)$ holds *and* the value of X is greater than three. Furthermore, constraints serve also as answers returned by derivations. For instance, if we add $q(X) \leftarrow \{X > 5\}$ to the clause above, then the goal $p(X)$ succeeds with answer $\{X > 5\}$. Of course, the standard resolution has to be extended in order to embed calls to the external solvers. At each resolution step new constraints are generated and collected, and the solver checks that the whole set of collected constraints is still satisfiable before execution can proceed further.

3 Expressive type inference with coinductive CLP(X)

In this section we define a constraint domain where constraint predicates are syntactic equality and subtyping over coinductive object and union types; we provide some simple examples of abstract compilation and of type inference obtained as resolution of specific goals.

An example of type domain The terms of our type domain are class, method and field names (represented by constants), and types coinductively defined over integer, boolean, object and union types.

$$\begin{aligned} bt &::= int \mid bool \\ t &::= bt \mid obj(c, [f_1{:}t_1, \ldots, f_n{:}t_n]) \mid t_1 \vee t_2 \end{aligned}$$

An object type $obj(c, [f_1{:}t_1, \ldots, f_n{:}t_n])$ specifies the class c to which the object belongs, together with the set of available fields with their corresponding types. The class name is needed for typing method invocations. We assume that fields in an object type are finite, distinct and that their order is immaterial. Union types $t_1 \vee t_2$ have the standard meaning [6, 9].

As pointed out in Section 2, in coinductive logic programming terms and derivations can correspond to infinite trees [7], hence not all the terms and derivations can be represented in a finite way, therefore the corresponding type systems defined in are called *idealized*. However, to be able to implement a sound approximation of an idealized type system one can restrict terms and derivations to regular ones. A regular tree can be infinite, but can only contain a finite number of subtrees or, equivalently, can be represented as the solution of a unification problem, that is, a finite set of syntactic equations of the form

$$(\text{bool})\frac{}{b \in bool} \qquad (\text{int})\frac{}{i \in int} \qquad (\vee\text{L})\frac{v \in t_1}{v \in t_1 \vee t_2} \qquad (\vee\text{R})\frac{v \in t_2}{v \in t_1 \vee t_2}$$

$$(\text{obj})\frac{v_1 \in t_1, \ldots, v_n \in t_n}{obj(c, [f_1 \mapsto v_1, \ldots, f_n \mapsto v_n, \ldots]) \in obj(c, [f_1{:}t_1, \ldots, f_n{:}t_n])}$$

Fig. 2. Rules defining membership

$X_i = e_i$, where all variables X_i are distinct and expressions e_i may only contain variables X_i [7, 17, 16].

A *type domain* is a constraint domain which defines two predicates: strong equivalence and subtyping. In this example strong equivalence corresponds to syntactic equality and is interpreted in the coinductive Herbrand universe, whereas subtyping is interpreted as set inclusion in a fixed type domain where types are interpreted as sets of values: $t_1 \leq t_2$ iff $[\![t_1]\!] \subseteq [\![t_2]\!]$, where $[\![t]\!]$ depends on the considered type language. In this section we consider the interpretation of union and object types as given by Ancona and Lagorio [2].

Values are coinductively defined by the following syntactic rules.

$$v ::= b \mid i \mid obj(c, [f_1 \mapsto v_1, \ldots, f_n \mapsto v_n]) \qquad (b \in \{fv, tv\}, i \in \mathbb{Z})$$

Fields in object values are finite and distinct, and their order is immaterial. Regular values correspond to finite, but cyclic, objects.

Interpretation is defined in terms of the judgment $v \in t$ coinductively defined by the rules of Figure 2.

All rules are intuitive. Note that an object value is allowed to belong to an object type having fewer fields; this is expressed by the ellipsis at the end of the value in the membership rule (obj). Not all infinite derivations can be considered valid, but only those that are *contractive* (see the definition below). In particular rules ($\vee$L) and ($\vee$R) are not contractive, hence cannot be consecutively applied an infinite number of times otherwise one could easily derive undesired judgments such as $fv \in t$ where $t = t \vee int$.

Definition 1. *A derivation for $v \in t$ is contractive iff it contains no subderivations built only with membership rules ($\vee R$), and ($\vee L$). The judgment $v \in t$ is derivable iff there is a contractive derivation for it.*

The interpretation $[\![t]\!]$ of type t is defined by $\{v \mid v \in t \text{ derivable}\}$.

An example of abstract compilation We present now a simple example of abstract compilation of a program into a Horn formula. Then, we give an example of infinite but not regular derivation in co-LP, and then show that such a derivation can be made regular when the subtyping constraint is considered, that is, when we shift from co-LP to co-CLP(X).

Consider the following class declarations written in Java-like syntax[5]. For making the example simpler we have omitted all type annotations and the declaration of a common implemented interface, since in this case type inference

[5] For simplicity, we restrict ourselves to a purely functional language.

works equally well without type annotations. However, we have already shown [1] how abstract compilation integrates pretty well nominal type annotations with inferred structural types.

```
class Zero { }

class Succ {
  pred;
  Succ(pred){ super(); this.pred=pred; }
}

class NatFact{
  create(i,n) { if (i<1)  return n;
                else  return create(i-1,new Succ(n)); }
}
```

Classes `Zero` and `Succ` encode the natural numbers with objects, while `NatFact` is a factory[6] class for objects encoding natural numbers.

To compile the program into a Horn formula, we introduce a predicate for each language construct; for instance, *invoke* for method invocation, *new* for constructor invocation, *field_acc* for field access, and *cond* for conditional expressions. Furthermore, auxiliary predicates are introduced for expressing the semantics of the language; for instance, predicate *has_meth* corresponds to method look-up. Each method declaration is abstractly compiled into a Horn clause: the compilation of method `create` generates the following clause.

```
has_meth(natFact,create,[This,I,N],R) ←
   rel_op(I,int,B),arth_op(I,int,I2),new(succ,[N],SN),
   invoke(This,create,[I2,SN],N2),cond(B,N,N2,R).
```

Method *has_meth* has four arguments: the class where the method should be found, the name of the method, the types of the arguments, including the type of the target object **this** which is always the first argument, and the type of the returned value. Each atom in the body of the clause corresponds to the abstract compilation of a sub-expression of the body of the method:

- `rel_op(I,int,B)` corresponds to `i<1` and asserts that `I op k` evaluates to `B` when `op` is a relational operator on integers, and `k` is any integer. Clearly, this means that `B` will be instantiated with type *bool*, and that `I` is required to be *int*, hence the clause above can be optimized as shown below.
- `arth_op(I,int,I2)` corresponds to `i-1` and asserts that `I op k` evaluates to `I2` when `op` is an arithmetic operator on integers, and `k` is any integer. Again, this atom can be easily optimized away as shown below.
- `new(succ,[N],SN)` corresponds to **new** `Succ(n)` and asserts that such an expression evaluates to `SN`.
- `invoke(This,create,[I2,SN],N2)` corresponds to **this**`.create(i-1,new Succ(n))` and asserts that such an expression evaluates to `N2`.

[6] The purpose of this simple example is just showing how abstract compilation to coinductive CLP(X) works.

– `cond(B,N,N2,R)` corresponds to the top expression which is the body of the method and asserts that it evaluates to `R`.

Additionally, one could add to the body of the clause the atom `inst_of(This, natFact)` (omitted here for simplicity) corresponding to the extra information that **this instanceof** `natFact` is necessarily true [1]. In the following, for simplicity we consider the optimized but equivalent clause:

```
has_meth(natFact,create,[This,int,N],R) ←
    new(succ,[N],SN),invoke(This,create,[int,SN],N2),
    cond(bool,N,N2,R).
```

Each method declaration is compiled into a clause defining predicate *has_meth*, and, analogously, each constructor declaration is compiled into a clause defining predicate *new*. Furthermore, other program independent clauses are generated to specify the behavior of the various constructs w.r.t. the available types. For instance, predicates *invoke* and *cond* are defined as follows:

```
invoke(obj(C,R),M,A,RT) ← has_meth(C,M,[obj(C,R)|A],RT).
invoke(T1∨T2,M,A,RT1∨RT2) ← invoke(T1,M,A,RT1),
                             invoke(T2,M,A,RT2).
cond(bool,T1,T2,T1∨T2).
```

Derivations and subtyping Let us consider an example of coinductive derivation for the following goal:

$$invoke(obj(natFact,[\,]),create,[int,obj(zero,[\,])],T_0).$$

We first observe that $new(succ,[N],obj(succ,[pred{:}N]))$ is derivable (for space reasons we have omitted the clauses for *new*); if we use the syntactic abbreviations $z = obj(zero,[\,])$, $s(z) = obj(succ,[pred{:}z])$, $s^n(z) = obj(succ,[pred{:}s^{n-1}(z)])$, $nf = obj(natFact,[\,])$ then $invoke(nf,create,[int,s(z)],T_1)$, $cond(bool,z,T_1,T_0)$ is obtained after some steps. With a resolution step, such a resolvent yields $invoke(nf,create,[int,s(z)],T_1)$ with $T_0 = z \vee T_1$. Therefore we obtain the following infinite non regular derivation (where the conclusion is at the bottom):

$$\begin{array}{cc}
\vdots & \vdots \\
invoke(nf,create,[int,s^n(z)],T_n) & T_{n-1} = s^{n-1}(z) \vee T_n \\
\vdots & \vdots \\
invoke(nf,create,[int,s(z)],T_1) & T_0 = z \vee T_1 \\
\vdots & \vdots \\
invoke(nf,create,[int,z],T_0) &
\end{array}$$

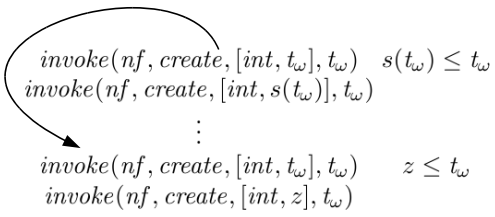
A possible solution is $T_0 = t_{\mathbb{N}}$, where $t_{\mathbb{N}}$ is the infinite non regular term $z \vee s(z) \vee \ldots \vee s^n(z) \vee \ldots$ which corresponds to the set of all objects encoding natural numbers. Clearly this result is purely theoretical, since neither the inferred type nor the derivation can be finitely represented. However, if we exploit the subtyping constraint $\leq$, then we can find a regular solution for the goal above with

a regular derivation. The key point is that each predicate is expected to behave in a specific way w.r.t. subtyping. If p is a predicate with only one[7] argument, we have the following four possibilities:

- p is *covariant* in its argument: if $p(t_1)$ and $t_1 \leq t_2$ hold, then $p(t_2)$ holds has well (we say that $p(t_1)$ subsumes $p(t_2)$).
- p is *contravariant* in its argument: if $t_1 \geq t_2$ (or, equivalently, $t_2 \leq t_1$) holds, then $p(t_1)$ subsumes $p(t_2)$.
- p is *weakly invariant* in its argument: if $t_1 \leq t_2, t_1 \geq t_2$ holds, then $p(t_1)$ subsumes $p(t_2)$. In this case we abbreviate $t_1 \leq t_2, t_1 \geq t_2$ with $t_1 \cong t_2$, and we call $\cong$ *weak equivalence*.
- p is *strongly invariant* in its argument: if $t_1 \equiv t_2$ holds, then $p(t_1)$ subsumes $p(t_2)$. We call $\equiv$ *strong equivalence* since it is expected to be stronger than $\cong$, that is $t_1 \equiv t_2 \Rightarrow t_1 \cong t_2$, but not conversely. In most cases $\equiv$ coincides with syntactic equality.

For instance, *invoke* is strongly invariant w.r.t. its first[8] and second arguments, contravariant in its third argument, and covariant in its fourth argument.

Coming back to our example, if t_ω is the regular term defined by $t_\omega = z \vee s(t_\omega)$, then we can build a regular derivation for the goal $\mathit{invoke}(\mathit{nf}, \mathit{create}, [\mathit{int}, z], t_\omega)$, since $\mathit{invoke}(\mathit{nf}, \mathit{create}, [\mathit{int}, t], u)$ subsumes $\mathit{invoke}(\mathit{nf}, \mathit{create}, [\mathit{int}, t'], u')$ if $\{t' \leq t, u \leq u'\}$ is satisfiable.



In the derivation we can apply two subsumption steps, since both constraints $z \leq t_\omega$ and $s(t_\omega) \leq t_\omega$ are satisfiable. The arrow indicates that the derivation is regular, since the atom $\mathit{invoke}(\mathit{nf}, \mathit{create}, [\mathit{int}, s(t_\omega)], t_\omega)$ is already present in the lower part of the derivation (that is, a coinductive hypothesis step can be applied).

[7] The definition can be easily generalized for an arbitrary number of arguments.

[8] In contrast with what intuition may suggest, weak invariance in the first argument of *invoke* is unsound since it allows non contractive derivations.

If we start with a non ground atom $invoke(nf, create, [int, z], T_0)$ we obtain the following incomplete derivation

$$\begin{array}{ll} invoke(nf, create, [int, s(X_2)], T_2) & s(X_1) \leq X_2, s(X_1) \vee T_2 \leq T_1 \\ \vdots & \vdots \\ invoke(nf, create, [int, s(X_1)], T_1) & z \leq X_1, z \vee T_1 \leq T_0 \\ \vdots & \vdots \\ invoke(nf, create, [int, z], T_0) & \end{array}$$

that can be transformed into a regular one, since $invoke(nf, create, [int, s(X_2)], T_2)$ can be derived from the coinductive hypothesis $invoke(nf, create, [int, s(X_1)], T_1)$; indeed, if we add the constraints $\{s(X_2) \leq s(X_1), T_1 \leq T_2\}$ to the current set of constraints, we get the set

$$\{z \leq X_1, z \vee T_1 \leq T_0, s(X_1) \leq X_2, s(X_1) \vee T_2 \leq T_1, s(X_2) \leq s(X_1), T_1 \leq T_2\}$$

which has solution $X_1 = X_2 = t_\omega$, $T_1 = T_2 = s(t_\omega)$, $T_0 \geq t_\omega$; hence, providing that we have an appropriate constraint solver, we can build a regular derivation returning a final set of constraints which is satisfied by a substitution mapping variables to regular terms. Note that another solution is given by $X_1 = X_2 = t_{\mathbb{N}}$, $T_1 = T_2 = s(t_{\mathbb{N}})$, $T_0 \geq t_{\mathbb{N}}$, and that t_ω is weakly equivalent to $t_{\mathbb{N}} \vee \omega$, where $\omega = s(\omega)$ (that is, the infinite ordinal ω), hence $t_{\mathbb{N}} \leq t_\omega$.

When type inference meets verification We have shown that the combination of regular, union and object types in conjunction with the subtyping relation used as a constraint allows inference of quite precise types; for instance, in the previous example we have shown that it is possible to infer that method `create` returns all objects implementing natural numbers.

However, it is possible to introduce more expressive types describing simple properties in form of data structure invariants of the program. Let us consider, for instance, a simple implementation of linked lists with two classes `EList` and `NEList` containing fields `e` for the first element, and `n` for the next list. Then, one might want to introduce the dependent type [19, 13] `list(`a`)` (where a is an arithmetic expression) describing all lists having length n, if a evaluates to n. In doing so, the subtyping relation has to be extended (and, consequently, the constraint solver) to take into account the new type. The interpretation of the type is defined by the following three new rules (where (lst-ev) is not contractive).

$$\text{(lst-0)}\frac{v \in obj(eList, [\,])}{v \in list(0)} \qquad \text{(lst-i)}\frac{v \in obj(neList, [e{:}\top, n{:}list(i-1)])}{v \in list(i)}\ i > 0$$

$$\text{(lst-ev)}\frac{v \in list(i)}{v \in list(a)}\ a = i$$

We use the top type $\top$ since we are interested in the length of the list, rather than in the type of its elements. Therefore, $\{list(0) \cong obj(eList, [\,]), list(i) \cong obj(neList, [e : \top, n : list(i-1)]), i > 0\}$ is satisfiable.

Let us consider the following more involved example, by introducing the type $avl(n)$ of the AVL trees of height n, implemented by the two classes `ETree` and `NETree` containing fields `e` (element at the root), `l` (left subtree), and `r` (right subtree). We recall that in AVL trees the heights of the two child subtrees of any node differ by at most one [15]. The interpretation of the type is defined by the following rules (where (avl-ev) is not contractive):

$$\text{(avl-0)}\frac{v \in obj(eTree, [\,])}{v \in avl(0)} \qquad \text{(avl-1)}\frac{v \in obj(neTree, [e{:}\top, l{:}obj(eTree, [\,]), r{:}obj(eTree, [\,])])}{v \in avl(1)}$$

$$\text{(avl-i)}\frac{\begin{array}{c} v \in \ obj(neList, [e:\top, l: avl(i-1), r: avl(i-1) \vee avl(i-2)]) \vee \\ obj(neList, [e:\top, l: avl(i-1) \vee avl(i-2), r: avl(i-1)]) \end{array}}{v \in avl(i)}\ i > 1$$

$$\text{(avl-ev)}\frac{v \in avl(i)}{v \in avl(a)}\ a = i$$

4 A prototype implementation of coinductive CLP(X)

In this section we show a prototype implementation of coinductive CLP(X), where the parameter X must be instantiated with a type domain $\mathcal{D}$. A type domain $\mathcal{D}$ defines two constraint predicates corresponding to strong equivalence $\equiv$ and subtyping $\leq$, as well as variance annotations for all the user-defined predicates. As we will see, variance annotations are more than a convenient syntactic notation for avoiding explicit insertion of constraints in the body of clauses; indeed, they allow definition of constraints which are associated with predicates, rather than clauses. To our knowledge, this is a novel feature not previously considered in CLP. In this way, we gain in expressive power, since instead of unification, subsumption can be exploited when the coinductive hypothesis rule is applied.

We first provide the fixed-point and operational semantics of coinductive CLP(X) programs defined over a type domain X.

Fixed-point semantics For simplicity, all following definitions use a fixed coinductive Herbrand universe and base. The notation $\overline{M}^n$ is a shortcut for $M_1, \ldots, M_n$, for any kind of meta-variables M_i.

We write $p_{\overline{\alpha}^n}$ to mean that predicate symbol p has arity n and variance annotation $\overline{\alpha}^n$, where each α_i may be one of the following constraint predicates $\{\leq, \geq, \cong, \equiv\}$, corresponding to covariance, contravariance, and weak and strong equivalence, respectively. As already explained in Section 3, $\geq$ and $\cong$ are derived constraints: $t_1 \geq t_2$ iff $t_2 \leq t_1$, and $t_1 \cong t_2$ iff $t_1 \leq t_2, t_2 \leq t_1$.

Definition 2. *If $p_{\overline{\alpha}^n}$ is a predicate symbol, then the ground atom $p_{\overline{\alpha}^n}(t_1, \ldots, t_n)$ subsumes the ground atom $p_{\overline{\alpha}^n}(t'_1, \ldots, t'_n)$ iff $\{t_1\alpha_1 t'_1, \ldots, t_n\alpha_n t'_n\}$ is satisfiable, that is $\mathcal{D} \models \{t_1\alpha_1 t'_1, \ldots, t_n\alpha_n t'_n\}$.*

$$\text{(empty)}\quad Hf \mid H \vdash \mathit{true} \rightsquigarrow \emptyset$$

$$\text{(co-hyp)}\frac{Hf \mid H_1, p_{\overline{\alpha}^n}(\overline{t}^n), H_2 \vdash G_1, G_2 \rightsquigarrow C_1 \qquad \vdash C_1 \cup C_2 \to C'}{Hf \mid H_1, p_{\overline{\alpha}^n}(\overline{t}^n), H_2 \vdash G_1, p_{\overline{\alpha}^n}(\overline{u}^n), G_2 \rightsquigarrow C'}\ C_2 = \mathit{gen}(\overline{t}^n, \overline{\alpha}^n, \overline{u}^n)$$

$$\text{(cls)}\frac{Hf \mid H, p_{\overline{\alpha}^n}(\overline{t}^n) \vdash G_1, G, G_2 \rightsquigarrow C_1 \qquad \vdash C_1 \cup C_2 \to C'}{Hf \mid H \vdash G_1, p_{\overline{\alpha}^n}(\overline{u}^n), G_2 \rightsquigarrow C'}\ \begin{array}{l} p_{\overline{\alpha}^n}(\overline{t}^n) \leftarrow G \text{ fresh instance of} \\ \text{a clause of } Hf \\ C_2 = \mathit{gen}(\overline{t}^n, \overline{\alpha}^n, \overline{u}^n) \end{array}$$

Fig. 3. Operational semantics

The one-step consequence function $T_{Hf,\mathcal{D}}$, induced by a Horn formula Hf where all predicates are associated with a variance annotation, is the function over sets of ground atoms contained in the coinductive Herbrand base, defined as follows:

$$T_{Hf,\mathcal{D}}(S) = \{A' \mid A \leftarrow A_1, \ldots, A_n \text{ ground instance of a clause in } Hf, \\ A_i \in S \text{ for all } i = 1, \ldots, n,\ A \text{ subsumes } A'\}$$

The coinductive Herbrand model of Hf w.r.t. the type domain $\mathcal{D}$ is the greatest fixed-point of $T_{Hf,\mathcal{D}}$. Equivalently, the semantics of Hf can be expressed by translating Hf into a formula Hf' where constraints are explicitly introduced in the clauses of Hf, and then by considering the greatest fixed-point of $T^{CLP}_{Hf',\mathcal{D}}$, where $T^{CLP}_{Hf',\mathcal{D}}$ is the standard one-step consequence function defined for CLP [11]:

$$T^{CLP}_{Hf,\mathcal{D}}(S) = \{A \mid A \leftarrow C, A_1, \ldots, A_n \text{ ground instance of a clause in } Hf, \\ A_i \in S \text{ for all } i = 1, \ldots, n,\ \mathcal{D} \models C\}$$

A clause having general shape $p_{\overline{\alpha}^n}(\overline{t}^n) \leftarrow \overline{A}^k$ is translated in the CLP clause $p_{\overline{\alpha}^n}(\overline{X}^n) \leftarrow \mathit{gen}(\overline{t}^n, \overline{\alpha}^n, \overline{X}^n), \overline{A}^k$, where $\overline{X}^n$ are distinct and fresh variables and constraints are generated by the function gen defined as follows:

$$\mathit{gen}(\epsilon, \epsilon, \epsilon) = \emptyset$$
$$\mathit{gen}((t, \overline{t}^{n-1}), (\alpha, \overline{\alpha}^{n-1}), (u, \overline{u}^{n-1})) = \{t\,\alpha\,u\} \cup \mathit{gen}(\overline{t}^{n-1}, \overline{\alpha}^{n-1}, \overline{u}^{n-1})$$

The function gen simply takes three tuples of the same length n, $t_1, \ldots, t_n$, $\alpha_1, \ldots, \alpha_n$, and $u_1, \ldots, u_n$, and generates the set of constraints $\{t_1\alpha_1 u_1, \ldots, t_n\alpha_n u_n\}$. This function is used in the next section for expressing the operational semantics of a Horn formula, where the meta-variables u_i may be instantiated with general terms and not only with variables.

Operational semantics The operational semantics of a Horn formula Hf w.r.t. a constraint domain $\mathcal{D}$ is inductively defined in Figure 3. The rules have been obtained as a synthesis of SLD resolution for coinductive LP and inductive CLP. When restricting to regular terms and proofs, results on the equivalence between the fixed-point and the operational semantics, which holds for both coinductive

LP [17] and inductive CLP [11], can be adapted also to the case of coinductive CLP.

The judgment $Hf \mid H \vdash G \rightsquigarrow C$ has the following meaning: the goal G succeeds w.r.t. the Horn formula Hf and the coinductive hypotheses H, and returns as solution the satisfiable set of constraints C. That is, as it happens for CLP, the solutions are all assignments of the variables in C to values that satisfy C. Clearly Hf, H, and G represent the input of the judgment, whereas the only output is C. The coinductive hypotheses H are needed for dealing with coinduction and hence for building regular derivations; for doing that, we have to keep track of all atoms that have been already resolved with a standard SLD step (see rule (cls) below); therefore, in practice, H is a stack of atoms.

The rules are parametric in the judgment $\vdash C \rightarrow C'$, which corresponds to the abstract specification of the constraint solver for the specific type domain under consideration, hence if the judgment is derivable then $\mathcal{D} \models C$ holds (hence, C represents the input of the solver) and returns an equivalent but simplified version C' (which, therefore, represents the output of the solver).

Rule (empty) deals with the empty goal (represented by *true*) which always succeeds; in this case the returned solution is the empty set of constraints.

Coinduction is managed by rule (co-hyp), where the atom $p_{\overline{\alpha}^n}(\overline{u}^n)$ (non deterministically selected from the goal) is resolved by using a coinductive hypothesis (non deterministically selected from H). This happens when H contains an atom $p_{\overline{\alpha}^n}(\overline{t}^n)$ (that is, with the same predicate symbol p and arity n of the atom selected form the goal) subsuming the atom $p_{\overline{\alpha}^n}(\overline{u}^n)$ of the goal for a certain assignment of values to variables. Such an assignment is determined by the set of constraints C_2 generated by $gen(\overline{t}^n, \overline{\alpha}^n, \overline{u}^n)$ and the set of constraints C_1 corresponding to the solution of the remaining atoms G_1, G_2 of the goal. Hence, if $C_1 \cup C_2$ is satisfiable for the solver, then the selected coinductive hypothesis subsumes the atom selected from the goal, and the rule is applicable. The returned solution is the simplification C' of $C_1 \cup C_2$ computed by the solver. Note that the rule uses subsumption instead of simple term unification, thanks to variance annotations. This would not be possible in standard CLP where constraints are associated with clauses and not with predicates.

Rule (cls) non deterministically selects an atom $p_{\overline{\alpha}^n}(\overline{u}^n)$ from the goal, and a clause from Hf s.t. its head has the same predicate symbol p and arity n of the atom selected from the goal. Then, an instance $p_{\overline{\alpha}^n}(\overline{t}^n) \leftarrow G$ of the clause where all variables are bijectively renamed with fresh variables is considered, and the new goal G_1, G, G_2, obtained by replacing the atom $p_{\overline{\alpha}^n}(\overline{u}^n)$ with the body G of the clause, is resolved w.r.t. the coinductive hypotheses augmented with the head $p_{\overline{\alpha}^n}(\overline{t}^n)$ of the clause. If resolution of G_1, G, G_2 succeeds with constraints C_1, and C_2 is the set of constraints generated from the head of the clause and the atom selected from the goal, then the solver checks whether $C_1 \cup C_2$ is satisfiable. If it so, then the clause is applicable, and resolution of the initial goal succeeds with the constraint set C' obtained by simplifying $C_1 \cup C_2$.

Prototype implementation We have implemented the operational semantics defined in Figure 3 with a meta-interpreter[9] written in SWI Prolog.

As it happens for Prolog interpreters, the implementation performs a depth first search of the tree containing all possible derivations, by selecting the atoms of the goal and the applicable clauses in the usual order (left to right and top to bottom, respectively). Furthermore, the interpreter first tries to apply coinductive hypotheses, hence rule (co-hyp) is tried prior than (cls); finally, coinductive hypotheses are selected starting from the top of the stack (that is, the most recent coinductive hypothesis is selected first). The basic structure of the meta-interpreter can be specified by the following pseudo-code.

```
coCLP(Goal, Solver, Solution) ←
  coCLP(Goal, Solver, [], [], Solution).
% (empty)
coCLP(true, _Solver, _CoHyp, Solution, Solution).
% (co-hyp)
coCLP((p_{ᾱ^n}(ū^n), Goal), Solver, CoHyp, C1, Solution) ←
  fresh_atom(p, n, p_{ᾱ^n}(X̄^n)),member(p_{ᾱ^n}(X̄^n), CoHyp),
  gen(X̄^n, ᾱ^n, ū^n, C2), union(C1, C2, C3), call(Solver, C3, C4),
  coCLP(Goal, Solver, CoHyp, C4, Solution).
% (cls)
coCLP((p_{ᾱ^n}(ū^n), Goal), Solver, CoHyp, C1, Solution) ←
  fresh_atom(p, n, p_{ᾱ^n}(X̄^n)), clause(p_{ᾱ^n}(X̄^n), Body),
  gen(X̄^n, ᾱ^n, ū^n, C2), union(C1, C2, C3), call(Solver, C3, C4),
  append_goal(Body, Goal, NewGoal),
  coCLP(NewGoal, Solver, [p_{ᾱ^n}(X̄^n)|CoHyp], C4, Solution).
```

The main predicate is `coCLP/3` (not specified in Figure 3), which is defined in terms of the auxiliary predicate `coCLP/5` which corresponds to the implementation of the judgment $Hf \mid H \vdash G \rightsquigarrow C$ specified in Figure 3. The definition is parametric in the predicate corresponding to the constraint solver, which is represented by the variable `Solver`. The two additional arguments of `coCLP/5` (when compared with `coCLP/3`) are the coinductive hypotheses and the accumulated constraints, which are both initially empty. The use of an accumulator for the generated constraints allows a more efficient implementation: `coCLP/5` is defined in terms of tail recursion, hence its execution can be optimized; furthermore, the constraints generated from the application of a coinductive hypothesis or of a clause are checked before proceeding with the resolution of the remaining atoms of the goal.

The search of an applicable coinductive hypothesis is performed by first creating an atom with the same predicate symbol and arity of the atom selected from the goal, where all arguments are fresh distinct variables (predicate `fresh_atom`, directly implementable with the standard meta-predicate **functor**), then such atom is searched in the list of coinductive hypotheses with the standard `member` predicate. Predicate `gen` corresponds to the function *gen* defined at the beginning of this section, whereas `union` performs union of sets of constraints.

[9] Available at `ftp://ftp.disi.unige.it/person/AnconaD/coCLP.zip`.

The implementation of rule (cls) (last clause) is analogous except for two details: the standard meta-predicate **clause** is used to find applicable clauses in the program, and the auxiliary predicate `append_goal` is used for appending the body of the selected clause to the remaining part of the initial goal.

Finally, we outline some of the details of our implementation not shown in the pseudo-code defined above.

To associate the variance annotation $\overline{\alpha}^n$ with the predicate p/n, one has to call the goal `register_coind_atom(`$p(\overline{\alpha}^n)$`)`. As a side effect of this call, every clause of p/n is associated with the set of constraints $gen(\overline{t}^n, \overline{\alpha}^n, \overline{X}^n)$, where $p(\overline{t}^n)$ is the head of the clause, and $\overline{X}^n$ are distinct and fresh variables. Hence, the meta-interpreter initially performs the same translation to an equivalent CLP program as defined at the beginning of this section. The set of pre-generated constraints is associated with the clause by means of a dynamically asserted fact containing the reference indexing the clause; such a reference can be retrieved with the library predicate **clause/3**.

Pre-generated constraints of clauses are exploited in two different ways: they are used for dynamically generating the set of constraints which must verified for allowing application of a clause, and for dynamically pre-generating the constraints corresponding to the variance annotation of the predicate of the coinductive hypothesis, which is pushed onto the stack when the clause turns out to be applicable. Such pre-generated constraints are used for dynamically generating the set of constraints which must verified for allowing the application of a coinductive hypothesis.

Finally, the answer returned by the interpreter is the set of computed constraints restricted to the variables contained in the initial goal; this final step has been not included in the pseudo-code described above.

5 Conclusion

We have proposed a novel general approach for expressing type inference as abstract compilation of programs into coinductive CLP(X), where X is a constraint domain defining strong type equivalence and subtyping. Furthermore, to the best of our knowledge, this is the first paper which formally defines the operational semantics of coinductive CLP and a presents an implementation based on a Prolog prototype.

Our approach seems particularly promising in the context of object-oriented programming, when the type domain contains union and object types. There are at least two interesting directions for investigating on the practicality of our approach. Devising a constraint solver for subtyping on regular union and object types is of paramount importance. We have already investigated its axiomatization [2, 3], and, based on this, we are devising an algorithm for implementing subtyping; however, we are not sure whether it is possible to implement a complete solver, since we still do not know whether subtyping on regular union and object types is decidable.

The other interesting direction of investigation concerns the ability of the approach to deal with imperative features; promising results have been already obtained in a recent work [4].

References

1. D. Ancona and G. Lagorio. Coinductive type systems for object-oriented languages. In S. Drossopoulou, editor, *ECOOP 2009*, volume 5653 of *Lecture Notes in Computer Science*, pages 2–26. Springer, 2009. Best paper prize.
2. D. Ancona and G. Lagorio. Coinductive subtyping for abstract compilation of object-oriented languages into Horn formulas. In *GandALF 2010*, Electronic Proceedings in Theoretical Computer Science, 2010. To appear.
3. D. Ancona and G. Lagorio. Complete coinductive subtyping for abstract compilation of object-oriented languages. In *FTfJP 2010*, ACM DL, 2010. To appear.
4. D. Ancona and G. Lagorio. Idealized coinductive type systems for imperative object-oriented programs. Technical report, DISI, January 2010. Submitted for journal publication.
5. D. Ancona, G. Lagorio, and E. Zucca. Type inference by coinductive logic programming. In *Post-Proceedings of TYPES'08*, number 5497 in Lecture Notes in Computer Science. Springer, 2009.
6. F. Barbanera, M. Dezani-Cincaglini, and U. de'Liguoro. Intersection and union types: Syntax and semantics. *Information and Computation*, 119(2):202–230, 1995.
7. B. Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25:95–169, 1983.
8. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13:451–490, 1991.
9. A. Igarashi and H. Nagira. Union types for object-oriented programming. *Journ. of Object Technology*, 6(2):47–68, 2007.
10. Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.
11. J. Jaffar and M.J. Maher. Constraint logic programming: A survey. *J. Log. Program.*, 19/20:503–581, 1994.
12. N.Saeedloei and G. Gupta. Verifying complex continuous real-time systems with coinductive CLP(R). In *Proc. of LATA 2010*, Lecture Notes in Computer Science. Springer, 2010. To appear.
13. N. Nystrom, V. A. Saraswat, J. Palsberg, and C. Grothoff. Constrained types for object-oriented languages. In *OOPSLA 2008*, pages 457–474, 2008.
14. M. Odersky, M. Sulzmann, and M. Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999.
15. Robert Sedgewick. *Algorithms*. Addison-Wesley, 1983.
16. L. Simon, A. Bansal, A. Mallya, and G. Gupta. Co-logic programming: Extending logic programming with coinduction. In *ICALP 2007*, pages 472–483, 2007.
17. L. Simon, A. Mallya, A. Bansal, and G. Gupta. Coinductive logic programming. In *ICLP 2006*, pages 330–345, 2006.
18. M. Sulzmann and P. J. Stuckey. HM(X) type inference is CLP(X) solving. *Journ. of Functional Programming*, 18(2):251–283, 2008.
19. Hongwei Xi. Dependent ML: an approach to practical programming with dependent types. *Journal of Functional Programming*, 17(2):215–286, 2007.

Verification of Software Product Lines: Reducing the Effort with Delta-oriented Slicing and Proof Reuse

— Position paper —

Daniel Bruns[1], Vladimir Klebanov[1], and Ina Schaefer[2⋆]

[1] Karlsruhe Institute of Technology, 76128 Karlsruhe, Germany
{bruns, klebanov}@kit.edu
[2] Chalmers University of Technology, 421 96 Gothenburg, Sweden
schaefer@chalmers.se

Abstract. Software product lines (SPL) are a well-known approach to develop industry-size adaptable software systems. SPL are often used in domains where high-quality software is desirable; the overwhelming product diversity, however, remains a challenge for assuring correctness. We present our work in progress on reducing the deductive verification effort across different products of an SPL. On the specification side, our approach enriches the delta language for describing relations between products of an SPL with formal specifications. On the verification side, we combine program slicing and similarity-guided proof reuse to ease the verification process.

1 Introduction

In this paper, we present our work in progress on deductive verification of software product lines. A software product line (SPL) [21] is a set of software systems (called *products*) with well-defined commonalities and variabilities. SPL are often used in domains (e.g., communications, medical, transportation) where high-quality software is desirable; the overwhelming product diversity, however, remains a challenge for assuring correctness by any method.

Even without formal verification, the dimensions and complexity of product lines make it essential to model the relationships between products explicitly. One of the authors has been working on the software engineering aspects of modeling SPL [23, 24, 22]. This has resulted in a modeling approach called delta-oriented programming (Section 2). Our current effort aims to exploit the information available in an SPL model to reuse verification results obtained from verifying one product when considering another product. Where necessary, we enrich the model with semantical information (such as formal specifications, Section 3). Considering other possibilities to verify SPL that are more meta-level

⋆ This author has been supported by Deutsche Forschungsgemeinschaft (DFG) and by the EU project FP7-ICT-2007-3 HATS.

(like generic or partial proofs), we decided to go on with a more light-weight approach first.

The technology that we are using to exemplify our proposal is Java for programming, JML [16] for formal specifications and the KeY system [6] for deductive verification. Our ideas should also be applicable to related technologies though. In particular, we try to make no assumptions about the verification system:

- We support both ways that verification systems can treat method calls: using the method contract or inlining the implementation. Using the contract is inherently modular while inlining is not, but it still has its advantages. It is simple, does not force the developer to write "trivial" contracts for helper methods, and reduces the number of commitments that need to be updated as the code evolves.
- Our method is also completely parametric on how a verification system treats invariants. In the worst case, all methods in the program need to be verified to preserve every invariant, as the invariant vocabulary is (in general) unrestricted. In practice verification systems use criteria such as visibility, syntax and typing, assignable clauses or ownership to reduce the workload. We simply limit ourselves to saying that all *relevant* methods must be checked.

Our approach consists of two parts. First, we analyze the SPL model to determine which parts of the original product are unchanged in the new product and do not have to be verified again but fulfill the specifications by product construction. This analysis uses static slicing techniques (Section 4). It is a new development and not yet implemented.

Second, for the modified parts, we apply a previously-developed (and implemented) proof reuse technique based on the assumption of similarity between the two implementation variants. It uses a similarity measure that determines which proof steps from proofs for the original product can be used to establish the proof obligations for the new product. This is a light-weight technique based on proof replay rather than on proof generation by construction (Section 5).

We present related work in Section 6 and draw conclusions in Section 7.

2 Delta-oriented Programming of Software Product Lines

Delta-oriented programming [23, 24, 22] is a novel approach to implementing software product lines. Delta-oriented programming offers an expressive and flexible programming "meta-language". Its aim is to relax the restrictions of currently established SPL description formalisms such as feature-oriented programming [5], which originated from the concept of stepwise development [4]. For a more detailed comparison between delta-oriented and feature-oriented programming, the reader is referred to [23].

In delta-oriented programming, an SPL is implemented as a *core module* together with a set of *delta modules*. The core module contains a complete product for some valid feature configuration, which can be developed by conventional

single-application engineering techniques. Delta modules specify changes to be applied in order to implement other products.

The notation we use for Java programs constituting individual products is the following:

Definition 1. *A* program *is a set of class declarations (further called simply classes) and a binary inheritance relation on this set. We are primarily interested in the transitive closure of this relation $\sqsubset$ and the transitive reflexive closure $\sqsubseteq$. $A \sqsubset B$ means that the class A is below class B in the inheritance hierarchy. Abstract classes and interfaces are omitted in this paper for brevity.*

A class *is a set of field and method declarations (which are built up of names, types, parameters, bodies, etc., as appropriate in Java). If C is a class declaring a method with signature m, then we will refer to this particular implementation as $C{::}m$. Vice versa, we identify the method signature m with a set of classes in a product that declare a method with that signature: $C \in m$ iff $C{::}m \in C$.*

Modification operations used in delta modules so far are the following:

- adding/removing a class declaration C: $adds(C)$, $removes(C)$
- modifying class C by
  - adding/removing a field f: $adds(C{::}f), removes(C{::}f)$
  - adding/removing a method declaration m: $adds(C{::}m), removes(C{::}m)$
  - changing the direct superclass of C to C': $reparents(C, C')$

The variability of an SPL is defined by the feature set F. Valid member products of an SPL are given by the feature model $\mathcal{F} \subseteq 2^F$. Each product uniquely corresponds to a combination of features, also called *feature configuration*. In the following, we identify products and feature configurations in $\mathcal{F}$. Each delta module d contains an *application condition* φ_d (the **when** clause in concrete syntax), which is a propositional formula over the feature set F. The application conditions specify which delta modules are necessary for which features. For every pair of valid products $P_1, P_2 \in \mathcal{F}$, $\Delta(P_1, P_2)$ is the set of delta modules that have to be applied to the product P_1 in order to obtain a product P_2 with a different feature configuration.[3]

The original delta language proposal [23] demands a partial order on deltas to guarantee that the result of applying $\Delta(P_1, P_2)$ is unique, as well as certain other syntactical well-formedness conditions, which we are not concerned with here. We just briefly mention a type system [7] ensuring that all products derivable in a delta-oriented product line are type-safe. The advantage of the type system is that it allows typechecking the core and delta modules in isolation, such that a change in one delta module only requires rechecking this delta module.

Example 1. Our running example will be a delta-oriented product line of bank accounts inspired by [10]. Figure 1a shows the core module of this SPL with the basic Account class. Figure 1b shows the delta module `DInvestment` for

[3] This is a slight generalization of the original delta approach, where deltas could only be applied to the core product.

```
core Base {
    class Account extends Object {
        int balance;
        int bonus;
        void addBonus(int x){}
        void update(int x) {
            balance += x;
        }
    }
}
```

(a) Core module with basic `Account` class

```
delta DInvestment when Investment {
    modifies class Account {
        removes void addBonus(int x);
        adds void addBonus(int x) {
            bonus += x;
        }
        removes void update(int x);
        adds void update(int x){
            balance += x;
            if (x > 0) addBonus(x/2);
        }
    }
}
```

(b) Delta module for feature `Investment`

```
class Account extends Object {
    int balance;
    int bonus;
    void addBonus(int x){
        bonus += x;
    }
    void update(int x) {
        balance += x;
        if (x > 0) addBonus(x/2);
    }
}
```

(c) The result of delta module application medskip

Fig. 1. Example of a delta-oriented SPL.

activating the `Investment` feature, which accumulates a bonus for each deposit made. Figure 1c contains the result of applying the delta module to the core, which is, again, a conventional Java class.

Later on, we will also see the `Paycheck` feature adding the class `Employer` as a client of `Account`. ◊

3 Delta-oriented Formal Specification of Software Product Lines

We use the Java Modeling Language (JML) [16] for the formal specification of product properties. In this work, we concentrate on class invariants and method contracts with pre- and post-conditions. As JML specifications are written directly into Java source files as comments, it is possible to include them in the delta language introduced in Section 2. Core modules are specified just as conventional programs. An example of a core module with JML specifications can be seen in the first listing of Example 3 (missing preconditions default to `true`).

For delta modules, we extend the delta language with the following operations to manipulate specifications:

- adding an invariant to a class: $adds(C, I)$
- removing an invariant from a class: $removes(C, I)$
- adding a contract (pre-/post-condition pair) to a method: $adds(C::m, ct)$
- removing a contract from a method: $removes(C::m, ct)$

Note that we only consider pairs of exactly one pre- and post-condition to be added or removed together. In case one of them is trivial (i.e., `true`), it is omitted.

Example 2. Figure 2 shows the delta module `DInvestmentSpec` changing the specifications in class `Account`. It is applied for the same configurations as the code delta `DInvestment`, since it has the same application condition.[4] ◊

It should be noted that in general there is no concordance between code deltas and specification deltas for one product. It is perfectly conceivable to change the code without changing the specification or the other way round. However, there are (at least) the following exceptions where code changes influence the specification:

- Removing a class or a method induces the removal of attached specifications.
- JML enforces behavioral subtyping, i.e., subclasses inherit the specifications of the superclass. Changing the inheritance hierarchy, thus, also changes the specification.
- JML by default enforces non-nullness of fields, variables, etc. Adding a field of reference type to a class automatically creates an implicit invariant about this field.
- Changing a (pure) method changes the semantics of specifications using this method.

[4] In principle, it is possible to specify code and specification changes in the same delta module. The separation in this work is for presentation reasons.

```
delta DInvestmentSpec when Investment {
    modifies class Account {
        removes //@ ensures bonus == \old(bonus);
          from void addBonus(int x);
        adds //@ requires x >= 0;
             //@ ensures bonus == \old(bonus) + x;
          to void addBonus(int x);
}
```

Fig. 2. A specification delta adds and removes pre- and post-conditions from methods.

4 Delta-oriented Slicing

If a product P_2 is derived from the product P_1 by delta application, in general, both the implementation as well as the specification change. However, from the information available in the used delta modules, we can infer conservatively to a large extent which specification parts of the new product P_2 may still be considered valid (due to proofs done for P_1) and which parts have to be (re-)proven in order to establish the specified properties.

The latter parts we call a *delta-oriented slice*, and in this section we present a procedure we have developed for computing it. Slicing originated in program analysis [28, 26] as a technique answering the question which program statements influence the value of a given variable at a given point. Our algorithm answers the question which proofs are influenced by a delta module. The basis of the algorithm are the definitions of the Java language [11] and JML.

Figure 3 shows the main slicing algorithm. As the first step of the algorithm, we copy all (valid) proofs from product P_1 into product P_2 regardless of their validity for P_2. In this process we already weed out all the proofs where the vocabulary involved (be it code or specifications) is no longer present in the new product. In the resulting over-approximated set of proofs for the new product, our algorithm identifies the proofs that do not hold in the new context and marks them as invalid. These proofs have to be redone. The algorithm also identifies new proof obligations that have to be discharged in order to obtain a full set of proofs for the specifications of P_2.

This goal is achieved by carrying out steps 2–12, each corresponding to a kind of change operator that can occur in $\Delta(P_1, P_2)$. For the sake of the algorithm, we assume that this set contains just one delta module (i.e., we assume delta module composition). The algorithm currently considers only the structural change information available in the delta and does not take the content of the modified methods or specifications into account. For some of the algorithm steps, we need to determine when the implementation $C::m$ is potentially referenced by the method invocation expression `e.m()`. This subroutine is shown in Figure 4.

Example 3. (i) We return to the bank account example introduced in Section 2. The core product with the basic `Account` class now contains specifications (see

Input: A product P_1, a set of proofs for the product P_1, and the delta $\Delta(P_1, P_2)$.
Output: A set of valid proofs for the product $P_2 = P_1 + \Delta(P_1, P_2)$.

1. Copy all proofs from P_1 to P_2 (regardless of validity). Weed out all proofs where the vocabulary involved (code or specification) is no longer present.

The following steps refer to the content of the delta module $\Delta(P_1, P_2)$:

2. For each *adds*(C):
   (a) do *adds*$(C::f)$ for each $f \in C$
   (b) do *adds*$(C::m)$ for each $m \in C$
3. For each *removes*(C): do *removes*$(C::f)$ and *removes*$(C::m)$ for each $f, m \in C$
4. For each *adds*$(C::m)$:
   (a) invalidate all pre-existing proofs where m was inlined and $C::m$ would have been among potentially referenced implementations (see Figure 4)
   (b) proofs using the contracts for m remain valid
   (c) prove that $C::m$ satisfies all specifications of C (either stated directly or inherited), as well as all other invariants
5. For each *removes*$(C::m)$:
   (a) invalidate all pre-existing proofs where m was inlined and $C::m$ would have been among potentially referenced implementations (Figure 4)
   (b) proofs using the contracts for m remain valid
6. For each *adds*$(C::f)$:
   (a) find the set of method implementations M referring to $C::f$ in P_2
   (b) invalidate all pre-existing proofs about any $C'::m \in M$
   (c) invalidate all pre-existing proofs inlining any $C'::m \in M$
   (d) invalidate all pre-existing proofs of specifications referring to $C::f$ in P_2
7. For each *removes*$(C::f)$: same as step 6, but look for $C::f$ in P_1
8. For each *reparents*(C, C'):
   (a) invalidate all pre-existing proofs inlining any $C''::m$ with $C'' \sqsubseteq C$
   (b) contracts for methods in reparented classes remain valid unless the contract no longer exists (inherited contract)
   (c) invalidate proofs for specifications inherited from any class K with $\widetilde{C} \sqsubseteq K \sqsubset \widehat{C}$ where $\widehat{C}$ is the least common supertype of C' and the old direct supertype $\widetilde{C}$ of C (see Figure 5 for an illustration)
   (d) prove that all classes $C'' \sqsubseteq C$ satisfy the specifications inherited from new superclasses K with $C' \sqsubseteq K \sqsubset \widehat{C}$
9. For each *adds*$(C::m, ct)$
   (a) prove that the contract ct is fulfilled by all $C'::m$ with $C' \sqsubseteq C$
10. For each *removes*$(C::m, ct)$
    (a) invalidate all pre-existing proofs that use the contract ct
11. For each *adds*(C, I)
    (a) prove that the invariant I is fulfilled by all relevant implementations
12. For each *removes*(C, I)
    (a) invalidate all pre-existing proofs that assume the invariant I

Fig. 3. The delta-oriented slicing algorithm

When is the implementation $C::m$ potentially referenced by the method invocation expression $e.m()$? We discern three different method invocation modes, defining for each a starting point class S of method lookup. The relation of S and C determines the answer:

Instance or "virtual" mode. This is the most common mode. The target expression (of type S) references an object (it may be an implicit `this` reference), and the method is not declared static or private. This invocation mode requires dynamic binding.
- If $C \sqsubseteq S$, then "yes"
- If $S \sqsubseteq C$ and exists $S' \sqsubseteq S$ such that for all K with $S' \sqsubseteq K \sqsubset C$ holds $K \notin m$, then "yes" (cf. Figure 6).
- Otherwise, "no".

Static mode (m is declared static or private). In this case, no dynamic binding is performed. The implementation to invoke is determined in accordance with the declared static type S of `e`. If $C = S$ then "yes", otherwise "no".

Super mode (`e` is the keyword `super`). This mode is used to access the methods of the immediate superclass S (of the class containing the invocation expression `e.m()`).
- If $S \sqsubseteq C$ and for all K with $S \sqsubseteq K \sqsubset C$ holds $K \notin m$, then "yes".
- Otherwise, "no".

Fig. 4. Subroutine: When is a method implementation potentially referenced?

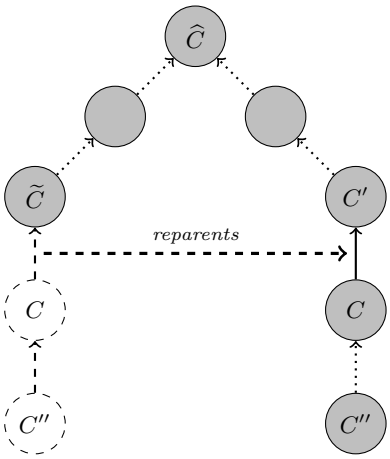


Fig. 5. Illustration of $reparents(C, C')$. Solid lines represent the direct subtyping relation, dotted lines its transitive closure, and dashed lines show relations of the previous product.

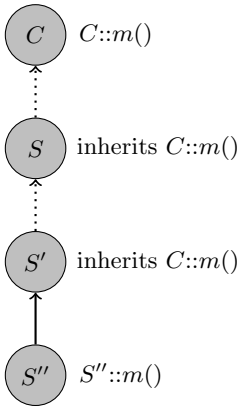


Fig. 6. Virtual method invocation mode and method overriding.

below). It can easily be proven that both methods satisfy their contracts and the class invariant.

```
core Base {
    class Account extends Object {
        //@ invariant bonus >= 0;
        int balance;
        int bonus;

        //@ ensures bonus == \old(bonus);
        void addBonus(int x){}

        /*@ ensures balance == \old(balance) + x;
          @         && bonus >= \old(bonus); @*/
        void update(int x) {
            balance += x;
        }
    }
}
```

(ii) Next, we apply the delta module shown below in order to generate a new product with the additional feature `Paycheck`. This module adds an `Employer` class with a reference to the account and a `payday()` method with a corresponding specification. In order to determine which proofs for the basic bank account are still valid, we use the delta-oriented slicing algorithm. We perform step 2 for the added class, leading to the steps 4 for the added method, 6 for the added field and 9 for the added contract. Only step 4c is non-trivial, since the method `payday()` did not exist before. The method can be verified easily – either by inlining the implementation of `addBonus()` and `update()` or by applying their contracts. There is no existing proof to reuse. Step 6 is trivial (the set M is empty) as the field `a` did not exist previously at all. Step 9 is subsumed by step 4 as `Employer` has no subclasses. No proofs are invalidated.

```
delta DPaycheck when Paycheck {
    adds class Employer extends Object {
        Account a;

        /*@ requires x >= 0 && bonus >= 0;
          @ ensures a.balance == \old(a.balance) + x
          @             && a.bonus >= \old(a.bonus);
          @*/
        void payday(int x, int bonus) {
            a.addBonus(bonus);
            a.update(x);
        }
    }
}
```

(iii) If we now want to incorporate the `Investment` feature as well, we apply the deltas `DInvestment` (Figure 1b) and `DInvestmentSpec` (Figure 2) to the latest product. These two deltas modify the implementation and specification of the method `addBonus()` and the implementation of the method `update()` in the class `Account`. The slicing steps to take to determine which proofs from the previous product are still valid are: 4 for the added methods, 5 for the removed methods, 9 for the added contract and 10 for the removed contract.

Steps 4c and 9 dictate that both `update()` and `addBonus()` have to be reproven for conformance with the class invariant and their respective (modified) contracts. Proof reuse is possible here (see Example 4 later). In contrast, `payday()` has not changed (neither code nor specification), but the proof that it satisfies its contract is now invalid. The proof has been invalidated by step 4a or 10, since it (the proof) depends on either the implementation or the contract of `addBonus()`. The proof reuse mechanism may be applied here to find a new proof efficiently. The contract of `update()` has not changed, and all proofs using it remain valid (step 4b). ♢

5 Proof Reuse for Modified Methods

In the delta-oriented slicing step, we have identified which specification parts in the newly generated product have to be rechecked. However, some method bodies that are modified may still have considerable similarities to the ones in the verified product. The correctness proofs of such modified methods are likely to resemble the old proofs. Here, we propose to employ proof reuse.

The reuse mechanism has been originally developed to save verification effort during incremental development (i.e., after fixing a bug). Since then the method has been applied to a number of different change management scenarios. The best account of proof reuse in KeY can be found in [14]. This part of our proposal is tailored to interactive verification systems like KeY, where the user provides hints to the prover by manipulating the proof object.

Of course, the proofs in our illustrating example are trivial and do not require proof reuse. In practice however, proofs do contain proof steps that cannot be (efficiently) found automatically. Users have to instantiate quantifiers, provide lemmas, loop invariants, and guide proof search in other ways. These efforts can be recycled by proof reuse. The main features of our reuse method are:

- The units of reuse are single rule applications. That is, proofs are reused incrementally, one proof step at a time. This allows us to keep our method flexible, avoiding the need to build knowledge about the target programming language or the particular calculus rules into the reuse mechanism. Furthermore, the soundness of proofs is guaranteed, since the usual rule application mechanism of the prover is used for proof construction.
- Proof steps can be adapted and reused even if the situation in the new proof is merely similar but not identical to the template.

– In case reuse has to stop because a part in the changed program is reached that requires genuinely new proof steps, reuse can be resumed later on when an unaffected part is reached. The system detects when this is the case.

The mechanism consists of two largely independent phases. Phase 1 analyzes the differences in the programs, identifies common code parts, and thus reusable subproofs (or rather beginnings of these subproofs). Phase 2 performs a similarity-guided replay of individual proof steps supported by the information gathered during phase 1.

For phase 1, we use the GNU `diff` utility, which is well-known to produce meaningful change sets for modifications to source files. We run `diff` on the source of the old and the new product (it is sufficient to limit the input to the changed method implementations in the delta scenario). Heuristics then help identify common code parts in both products based on `diff` output. The proof fragments dealing with these common parts are good candidates for reuse.

Example 4. This is `diff` output for the `update()` method between a product implementation not including the `Investment` feature and one including it:

```
     void update(int x) {
         balance += x;
+        if (x > 0) addBonus(x/2);
     }
```

The plus sign on the left in the output of the `diff` shows that the last line of the method was inserted in the new product, while everything else remained unchanged. We, thus, identify two reusable subproofs in the template proof for the old version of the `update()` method: one that starts with the beginning of the `update()` method and the other containing the proof steps after the method has ended. These are the points where reuse will restart after the inserted code calling the `addBonus()` method has been treated. ◊

During phase 2, the reuse mechanism performs proof replay. In contrast to other approaches, this form of proof replay is based not just on the structure of the proof, but also on a similarity measure between sequents involved in proof steps. The similarity measure indicates whether it is appropriate to reuse a particular proof step from the template proof in a given situation in the new proof.

The presented proof reuse technique works well for changed implementations; it can also be used for changed specifications but much less effectively. Specifications are less structured than programs, and proof shapes adhere to implementations rather than specifications, which makes finding reusable subproofs much harder.

6 Related Work

Formal methods are used in the context of software product lines for a variety of applications. A large body of work is concerned with the formal analysis of

feature models [2] or product models [17]. Further approaches (e.g., [9]) verify that the variability specified by a feature model is correctly implemented in code. There are several approaches to guarantee type-safety of feature-oriented product line implementations [1, 10, 25] by means of external analyses or type systems. Efficient verification of product behavior, however, is not well established. In testing [18, 20] or model checking [15, 8] there is work to make validation of product lines more efficient, though.

In [3], a case study for the product line development of a compiler is considered. The compiler is developed by stepwise refinement or extension of the compiler functionality. The correctness proof of the compiler is extended and refined inline with the functional extensions by introduction or adaptation of invariants and the addition of case distinctions. This approach relies on a fixed structure of the induction proof for compiler correctness that allows determining in advance which modifications of the proof are required by functional changes. In our work, we do not have a predetermined proof structure since we consider arbitrary behavioral properties formulated by JML specifications.

Reuse of verification artifacts can profit from the plethora of available insight in related fields, such as slicing for debugging [28, 26] or model checking [12], reuse of refined specifications [27], change management in theory development [19, 13], incremental compilation, refactoring, and software change impact analysis.

Of course, deriving a new product in a product line is also closely related to evolving a single product. Most verification systems implement some kind of proof management for this case. Alas, system developers apparently – and unjustifiedly, we think – tend to consider this important component an implementation detail, as published accounts on this subject are rare.

7 Conclusions

Working on verification of SPL we have come to a number of conclusions. Most of them regard transition from a syntactic to a more semantic-based modeling of SPL.

- In order to define delta operations on specifications in a meaningful way, it is necessary to uniquely identify class invariants and method contracts (e.g., for removal or modification). This could be handled by introducing labels (as most tools probably already do internally).
- So far the operations we have defined for specification deltas are rather basic. One reason for this is simplicity. The other one is that at least with the current calculi, the shape of a proof follows rather closely the shape of the program, but it is much less related to the shape of a specification. It remains to be seen whether adding more fine-grained change information in the specification deltas helps obtaining new proofs more efficiently. Additional operators that appear promising to us are case distinctions and redundant specifications (lemmas).
- Until now, the delta module operations (for code) and their applicability conditions are mostly syntactical. Greater power and precision can be achieved

by adding more semantical information. For instance, such a description might dictate that a certain feature is only compatible with another one if the base product preserves certain data invariants. New tools could be devised to assist in deriving consistent products with desired behavior based on semantical information.

– Getting the formal specification of a product right is difficult. Deriving a correct product from another is even more difficult. Even if two products P_1 and P_2 fulfill the specification I (as ensured by our approach), it is still only *syntactically* the same specification I. With current state of specification languages, it is hard to infer to which degree changes in implementation or other specifications between P_1 and P_2 may cause the *semantics* of I to change in subtle and unintended ways. Such a change constitutes a complex retrenchment that involves the complete semantics of the specification language. This issue needs further investigation.

References

1. Sven Apel, Christian Kästner, Armin Grösslinger, and Christian Lengauer. Type safety for feature-oriented product lines. *Automated Software Engineering An International Journal*, 2010.
2. Don S. Batory, David Benavides, and Antonio Ruiz-Cortes. Automated analysis of feature models: Challenges ahead. *Communications of the ACM*, 49(12), 2006.
3. Don S. Batory and Egon Börger. Modularizing theorems for software product lines: The Jbook case study. *Journal of Universal Computer Science*, 14(12), 2008.
4. Don S. Batory and Sean W. O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Trans. Softw. Eng. Methodol.*, 1(4):355–398, 1992.
5. Don S. Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling step-wise refinement. *IEEE Trans. Software Eng.*, 30(6):355–371, 2004.
6. Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *Lecture Notes in Computer Science*. Springer-Verlag, 2007.
7. Lorenzo Bettini, Ferruccio Damiani, and Ina Schaefer. FΔJ: A core calculus for delta-oriented programming, 2010. `http://www.di.unito.it/~damiani/papers/fdj.pdf`.
8. Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, Axel Legay, and Jean-Franois Raskin. Model checking lots of systems: Efficient verification of temporal properties in software product lines (to appear). In *32nd International Conference on Software Engineering, ICSE 2010, May 2-8, 2010, Cape Town, South Africa, Proceedings*. IEEE, 2010.
9. Krzysztof Czarnecki and Krzysztof Pietroszek. Verifying feature-based model templates against well-formedness OCL constraints. In *Conf. on Generative Programming and Component Engineering (GPCE)*, 2006.
10. Benjamin Delaware, William Cook, and Don Batory. A Machine-Checked Model of Safe Composition. In *Foundations of Aspect-Oriented Languages (FOAL)*, pages 31–35. ACM, 2009.
11. James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley Longman, Amsterdam, 3rd edition, 2005.

12. John Hatcliff, Matthew B. Dwyer, and Hongjun Zheng. Slicing software for model construction. *Higher-Order and Symbolic Computation*, 13(4):315–353, 2000.
13. Dieter Hutter. Management of change in structured verification. In *Automated Software Engineering (ASE)*, page 23, 2000.
14. Vladimir Klebanov. Proof reuse. In Beckert et al. [6].
15. Kim Lauenroth, Klaus Pohl, and Simon Toehning. Model checking of domain artifacts in product line engineering. In *Automated Software Engineering (ASE)*, pages 269–280. IEEE Computer Society, 2009.
16. Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: a behavioral interface specification language for Java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, 2006.
17. Mike Mannion. Using first-order logic for product line model validation. In Garry Chastek, editor, *Software Product Lines: Proceedings of the Second Software Product Line Conference (SPLC2)*, LNCS 2379, pages 176–187, San Diego, CA, August 2002. Springer.
18. John D. McGregor. Testing a software product line. Technical Report CMU/SEI-2001-TR-022, Software Engineering Institute, Carnegie Mellon University, December 2001.
19. Till Mossakowski. Heterogeneous theories and the heterogeneous tool set. In Yannis Kalfoglou, W. Marco Schorlemmer, Amit P. Sheth, Steffen Staab, and Michael Uschold, editors, *Semantic Interoperability and Integration*, volume 04391 of *Dagstuhl Seminar Proceedings*. IBFI, Schloss Dagstuhl, Germany, 2005.
20. Henry Muccini and André van der Hoek. Towards testing product line architectures. *Electr. Notes Theor. Comput. Sci*, 82(6), 2003.
21. Klaus Pohl, Günther Böckle, and Frank van der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer, Heidelberg, 2005.
22. Ina Schaefer. Variability modelling for model-driven development of software product lines. In *4th Int. Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, Linz, Austria, January 2010.
23. Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. Delta-oriented programming of software product lines. In *Proceedings, 14th International Software Product Line Conference*, Lecture Notes in Computer Science, Jeju, South Korea, 2010. Springer. To appear.
24. Ina Schaefer, Alexander Worret, and Arndt Poetzsch-Heffter. A model-based framework for automated product derivation. In *Model-driven Approaches in Software Product Line Engineering (MAPLE 2009)*, 2009.
25. Sahil Thaker, Don Batory, David Kitchin, and William Cook. Safe composition of product lines. In *GPCE*, pages 95–104. ACM, 2007.
26. Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3), 1995.
27. Heike Wehrheim. Slicing techniques for verification re-use. *Theor. Comput. Sci*, 343(3):509–528, 2005.
28. Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, August 1984.

Author Index