Bernhard Beckert
Claude Marché (Eds.)

LNCS 6528

Formal Verification of Object-Oriented Software

International Conference, FoVeOOS 2010
Paris, France, June 2010
Revised Selected Papers

Springer

Bernhard Beckert   Claude Marché (Eds.)

Formal Verification of Object-Oriented Software

International Conference, FoVeOOS 2010
Paris, France, June 28-30, 2010
Revised Selected Papers

Springer

Volume Editors

Bernhard Beckert
Institute for Theoretical Informatics
Am Fasanengarten 5, 76131 Karlsruhe, Germany
E-mail: beckert@kit.edu

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

Library of Congress Control Number: 2010941559

CR Subject Classification (1998): D.2.4, D.2, D.1.5, F.3, D.3, K.6, F.4

LNCS Sublibrary: SL 2 – Programming and Software Engineering

ISSN 0302-9743
ISBN-10 3-642-18069-8 Springer Berlin Heidelberg New York
ISBN-13 978-3-642-18069-9 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper 06/3180

Preface

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#. The International Conference on Formal Verification of Object-Oriented Software (FoVeOOS 2010) aimed to foster collaboration and interaction among researchers in this area. It was held during June 28–30, 2010 in Paris, France.

FoVeOOS was organized 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 Program Committee selected 23 for presentation at the conference. In addition to the contributed papers, the program 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.

This volume contains a selection of research papers and system descriptions presented at FoVeOOS 2010. Authors of the 23 papers presented at the conference[1] were invited to submit improved versions, to be reviewed a second time. Twenty-one submissions were received, and the Program Committee selected 11 of them. Additionally, two of the invited speakers provided papers, which were reviewed by the Program Committee and included in this volume.

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

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

October 2010

Bernhard Beckert
Claude Marché

[1] Proceedings containing all papers presented at the conference are available at http://digbib.ubka.uni-karlsruhe.de/volltexte/1000019083.

Organization

Program Committee

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

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

Table of Contents

From a Proven Correct Microkernel to Trustworthy Large Systems

June Andronick

NICTA*, UNSW
june.andronick@nicta.com.au

Abstract. The seL4 microkernel was the world's first general-purpose operating system kernel with a formal, machine-checked proof of correctness. The next big step in the challenge of building truly trustworthy systems is to provide a framework for developing secure systems on top of seL4. This paper first gives an overview of seL4's correctness proof, together with its main implications and assumptions, and then describes our approach to provide formal security guarantees for large, complex systems.

1 Introduction

The work presented here aims to tackle the general challenge of building truly trustworthy systems. The motivation is classic: software is ubiquitous and in use in systems that are more and more critical. This issue being well-accepted does not prevent the observation [4] that we routinely trust systems which again and again demonstrate their lack of trustworthiness.

The approach taken here follows the idea [10] of minimising the amount of code that need to be trusted, known as the *trusted computing base* (TCB), i.e. the part of the system that can potentially bypass security. What is added here is then to *prove* that this TCB can actually be trusted, *prove* that it is implemented in such a way that it does not bypass security. And by proving, we mean providing a formal, mathematical proof.

The first step in taking up this challenge has been to concentrate on the unavoidable part of the TCB: the operating system's core, its kernel. The kernel of a system 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 idea of minimising the TCB applied to kernels led to the concept of *microkernels*. 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

* 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.

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 [8], 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 general-purpose microkernel whose functional correctness has been formally proved: seL4 [6]. Section 2 gives an overview of this proof, its assumptions, results and implications, and overall effort.

Given this trustworthy foundation, we are now looking at designing and building large, complex systems, for which formal guarantees can be provided about their safety, security and reliability. Our vision, together with our on-going and future work, are described in Section 3.

2 A Proven Correct OS Kernel

The challenges in providing a "formally proven correct, general-purpose microkernel" are multiple, but all mainly come down to building a system that is both verifiable and suitable for real use. From the formal verification point of view, complexity is the enemy. From the kernel point of view, performance is the target. These two diverging objectives have been met by designing and implementing a new kernel, from scratch, with two teams working together and in parallel on formalisation and optimisations.

This kernel, called seL4, is a microkernel of the L4 family designed for practical deployment in embedded systems with high trustworthiness requirements. 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, enforced using the hardware's memory management unit (MMU). All memory, devices, and microkernel-provided services require an associated *capability* [3], i.e. an access right, to utilise them. The set of capabilities a component possesses determines what a component can directly access.

The formal verification work aimed at proving the kernel's *functional correctness*, i.e. proving that the kernel's implementation is correct with respect to a formal specification of its expected behaviour. Formally, we are showing *refinement*: all possible behaviours of the C implementation are a subset of the behaviours of the abstract specification. For this, we use interactive, machine-assisted and machine-checked proof, namely the theorem prover Isabelle/HOL [9].

In practice, this was done is several steps, as shown in Figure 1. First, increasingly complete prototypes of the kernel were developed in the functional language Haskell. On one hand, low-level design evaluation was enabled by a realistic execution environment that is binary-compatible with the real kernel. On the other hand, the Haskell prototype could be automatically translated in the theorem prover as the formal design specification, where the refinement to the abstract specification could be started. This first refinement step represents a proof that the design is correct with respect to the specification. Since the Haskell prototype did not satisfy our high-performance requirement, we then

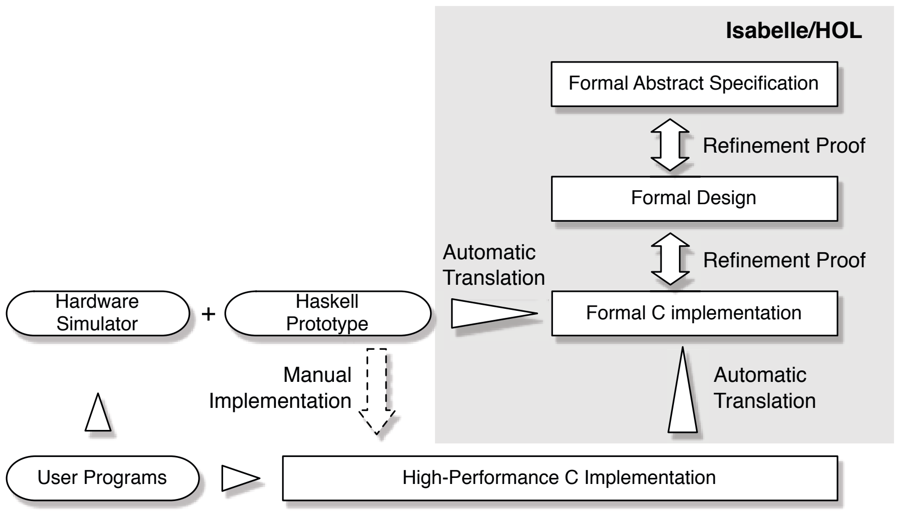


Fig. 1. The seL4 design process and refinement layers

manually translated it into high-performance C code, giving opportunities for micro-optimisations. The C code was then translated into Isabelle, using a very precise and faithful formal semantics for a large subset of the C programming language [6,11,12]. A second refinement step then proved that the C code, translated in Isabelle, was correct with respect to the formal design [13].

The refinement being transitive, the two refinement steps give us a formal proof that the C implementation of seL4 refines its formal 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 (there is also an unverified x86 port of seL4). Under those assumptions, the functional correctness proof also gives us mathematical proof that the seL4 kernel is free of buffer overflows, NULL pointer dereferences, memory leaks, and undefined execution. The verification revealed around 460 bugs, both on the design and implementation. The total effort amounted to 2.5 person years (py) to develop the kernel and 20 py for the verification, including 9 py invested in formal language frameworks, proof tools, proof automation, theorem prover extensions and libraries. More details about the assumptions, implications and effort can be found in [6,5].

The overall key benefit of a functional correctness proof is that proofs about the C implementation of the kernel can now be reduced to proofs about the specification for properties preserved by refinement. The correspondence established by the refinement proof ensures that all Hoare logic properties of the abstract model also hold for the refined model. This means that if a security property is proved in Hoare logic about the abstract model (not all security properties can be), our refinement guarantees that the same property holds for the kernel source code.

3 Trustworthy, Large Systems

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. Although verifying programs with sizes approaching 10 000 lines of code is a significant improvement in what formal methods were previously able to verify with reasonable effort, it still represents a significant limit on the verification of modern software systems, frequently consisting of millions of lines of code.

Our vision to verify such large and complex systems comes from the observation [1] that not all software in a large system necessarily contributes to a given property of interest. For instance, the user interface of a medical device might represent a large amount of code, and ideally, the safe delivery of medicine should not have to rely on it. Similarly, the entertainment system implementation in a car should not have any impact on the safety of the braking system.

The idea is thus again to minimise the TCB, minimise the amount of code which the desirable property relies on, to a size where formally verifying its exact behaviour is still possible. Formally proving that the property holds for the overall system then consists in proving that it holds for the trusted components, modelled by their expected behaviours, and proving that the untrusted parts are isolated, i.e. that nothing needs to be verified about them. The key here is to use seL4's access control mechanisms to enforce this isolation between the trusted and the untrusted parts: what untrusted components can access is determined by the set of capabilities they hold. Careful choice of initial capabilities distribution can thus isolate large parts of software to exclude them from the TCB.

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.

More precisely, Figure 2 illustrates the different steps our approach proposes. First, the architecture of the system defines the components needed for the system, and the capabilities they need to hold. This initial capabilities distribution defines the partition between trusted and untrusted components, with respect to a desired property for the system. We have defined a capability distribution language, called capDL [7], with a formal semantics that enables us to formally describe what the exact initial distribution is expected to be.

The next step is to prove that, given this initial capability distribution and the identified partition between trusted and untrusted components, the targeted property holds on the entire system. To avoid having to reason on the complex, detailed and low-level capDL description, we first abstract the architecture description into a simpler, high level security architecture. The aim is to have the abstraction done automatically, together with a formal proof of refinement. The

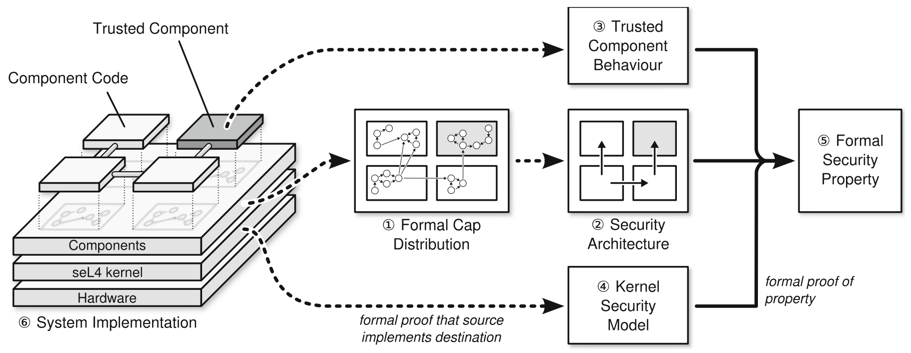


Fig. 2. Full-system verification approach for seL4-based system

property is then proved at this abstract level. The trusted components' behaviour is modelled as the sequence of kernel instructions they are expected to perform. At this abstract level, the kernel instructions are described in a high level security model of the kernel. The untrusted components' behaviour is modelled as any instruction authorized by the set of capabilities they hold. The concurrent execution of all components is modelled as all possible interleavings of instructions from any component in the system.

The proof of the property implicitly validates the identified partition between trusted and untrusted components: if the proof succeeds, it means that the property indeed does not depend on the untrusted components' behaviours, and that they will be correctly implemented by any concrete program code. In some cases, the property may not be proved, revealing some issues in the design that need to be fixed.

Inspired by seL4's successful "design for verification" process, we believe that the design and implementation of the components should be done in parallel in an iterative process. Although the implementation of the untrusted components is not constrained, the proof does depend on the trusted components' behaviour. Therefore, for the property to hold not only on the abstract level but on the actual implementation, the trusted components' code has to be shown correct with respect to the expected formal behaviour used for the proof. This would follow and use the refinement approach and framework developed for seL4 verification. Similarly, we need to prove that the initial boot code leads to a state satisfying the expected formal initial capability distribution. This is ongoing work. Finally, we need to prove that the kernel's code refines its security model used to model the trusted components instructions. Building on existing seL4 refinement layers (Figure 1), this comes down to adding a layer on the top of the stack and proving that the formal abstract specification refines the security model (with additional work to prove that seL4's access control mechanism indeed ensures isolation). All of this is ongoing work.

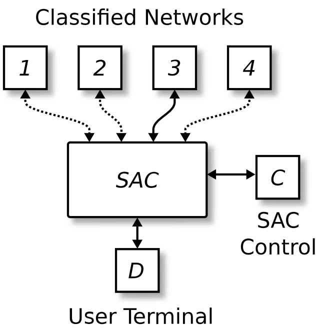


Fig. 3. The SAC routes between a user's terminal and 1 of n classified networks

The main gain in this vision is that formal guarantees can be made for a large complex system's *implementation*, while ignoring the identified large untrusted components, leaving only the trusted components to be formally verified.

The first steps of the approach have been demonstrated on a concrete example system, namely a multilevel secure access controller (SAC) aiming to isolate networked services of different classification levels, as illustrated in Figure 3. In this case study the user only needs to access one network at a time, and selects the network through a web interface provided by the SAC on a control network interface. The property the SAC must ensure is that all data from one network is isolated from each of the other networks. While we assume that the user's terminal is trusted to not leak previously received information back to another network, we otherwise assume that all networks connected to the SAC are malicious and will collude. The SAC is representative of systems with simple requirements, but involving large, complex components, here a secure web interface, network card drivers, a TCP/IP stack for the web server, and IP routing code, any one individually consisting of tens of thousands of lines of non-trivial code.

The architecture that has been designed for the SAC is represented in Figure 4, where the user's terminal is connected to NIC-D, while the SAC is controlled through a web interface provided on NIC-C, and for simplicity of explanation, we assume that the SAC only needs to multiplex two classified networks, NIC-A and NIC-B. The system's security architecture has been designed to minimise the TCB to a single trusted component (in addition to the underlying kernel): the router manager. The router manager is the only component with simultaneous access to both NIC-A and NIC-B. The aim is that it does never use those accesses (capabilities) to access NIC-A or NIC-B, but only holds them to grant one or the other to an untrusted router component in charge of routing between one network and the user terminal. Another untrusted component, the SAC controller, provides a web interface to the control network on NIC-C. When the SAC needs to switch between networks, the SAC controller informs the router manager, which deletes the running router component and sanitises the hardware registers and buffers of NIC-D (to prevent any residual information from inadvertently being stored in it). The router manager then recreates the router,

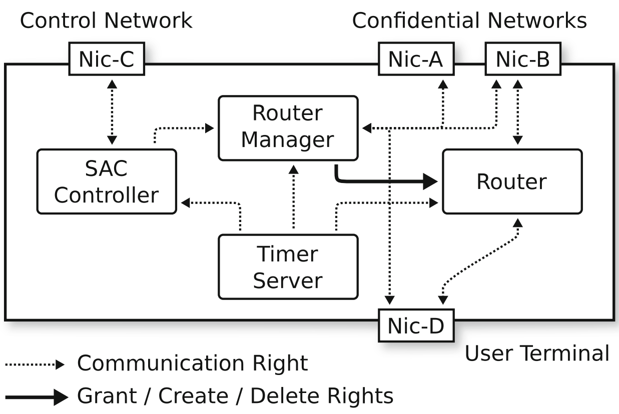


Fig. 4. High-level component breakdown of the SAC design. The router manager is the only trusted component in the system, as no other component has simultaneous access to both NIC-A and NIC-B.

and grant it access to NIC-D and either NIC-A or NIC-B as required. This allows the router to switch between NIC-A and NIC-B without being capable of leaking data between the two.

We therefore only need to trust the router manager's implementation (approximately 1500 lines of code) not to violate the isolating security policy, but can ignore the two other large untrusted components, that we implement as Linux instances, comprising millions lines of code. At least this is what we expect, we now have to prove it.

For this case study, we first formalised the low level design in capDL, leading to a detailed description of the initial capability distribution in terms of kernel objects, as shown in Figure 5. Then we manually abstracted this design into an abstract security architecture between high level components, as the one in Figure 4. Doing this step automatically, together with a proof of refinement, is part of our future work. Finally we have formally shown that with this security architecture, information cannot flow from one back-end network to another. Details of the proof can be found in [2].

What remains to be done for this case study is to prove that (1) the router manager's code refines its formal behaviour used for the proof; (2) the booting code leads to the state illustrated in Figure 5. We also need to prove the kernel's security model refinement to the code, which in this case would also involve extending our existing functional correctness proof to the x86 version of seL4 used for the case study.

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.

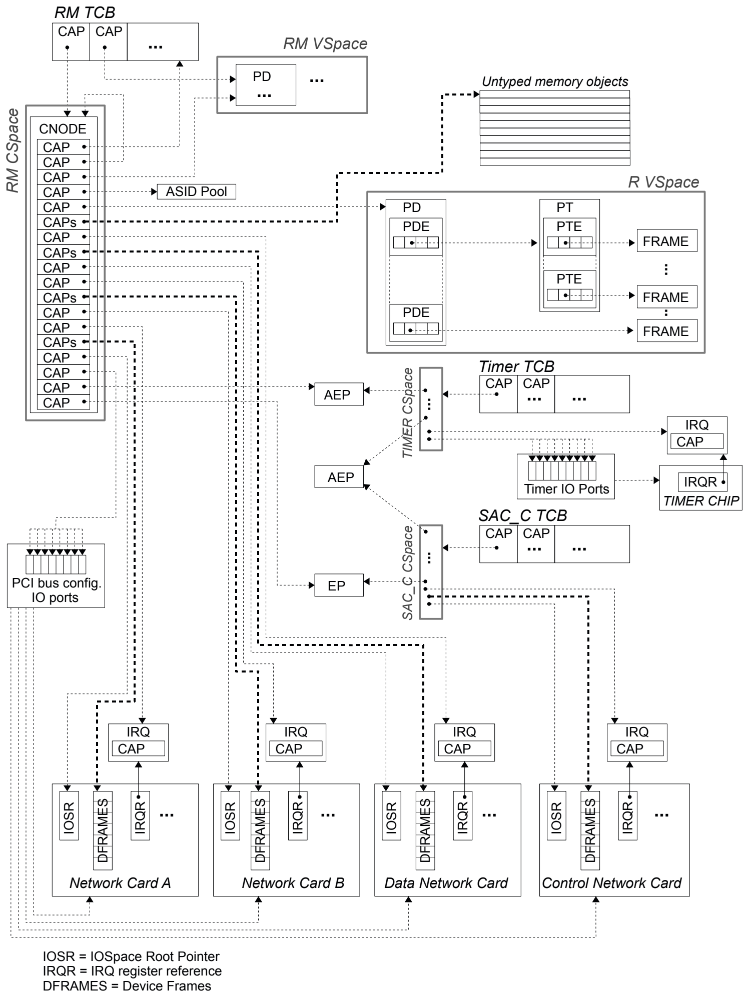


Fig. 5. Low-level Design

Acknowledgments. The proof of the SAC mentioned above was conducted almost entirely by David Greenaway with minor contributions from Xin Gao, Gerwin Klein, and myself. The following people have contributed to the verification and/or design and implementation of seL4 (in alphabetical order): June Andronick, Timothy Bourke, Andrew Boyton, David Cock, Jeremy Dawson, Philip Derrin Dhammika Elkaduwe, Kevin Elphinstone, Kai Engelhardt, Gernot Heiser, Gerwin Klein, Rafal Kolanski, Jia Meng, Catherine Menon, Michael Norrish, Thomas Sewell, David Tsai, Harvey Tuch, and Simon Winwood.

References

1. Alves-Foss, J., Oman, P.W., Taylor, C., Harrison, S.: The MILS architecture for high-assurance embedded systems. Int. J. Emb. Syst. 2, 239–247 (2006)
2. Andronick, J., Greenaway, D., Elphinstone, K.: Towards proving security in the presence of large untrusted components. In: Klein, G., Huuck, R., Schlich, B. (eds.) 5th SSV, Vancouver, Canada, USENIX (October 2010)
3. Dennis, J.B., Van Horn, E.C.: Programming semantics for multiprogrammed computations. CACM 9, 143–155 (1966)
4. Heiser, G., Andronick, J., Elphinstone, K., Klein, G., Kuz, I., Ryzhyk, L.: The road to trustworthy systems. In: 5th WS Scalable Trusted Comput., Chicago, IL, USA (October 2010)
5. Klein, G., Andronick, J., Elphinstone, K., Heiser, G., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: Formal verification of an OS kernel. CACM 53(6), 107–115 (2010)
6. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: Formal verification of an OS kernel. In: 22nd SOSP, Big Sky, MT, USA, pp. 207–220. ACM, New York (October 2009)
7. Kuz, I., Klein, G., Lewis, C., Walker, A.: capDL: A language for describing capability-based systems. In: 1st APSys, New Delhi, India (to appear, August 2010)
8. Liedtke, J.: Towards real microkernels. CACM 39(9), 70–77 (1996)
9. Nipkow, T., Paulson, L., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic. LNCS, vol. 2283. Springer, Heidelberg (2002)
10. Saltzer, J.H., Schroeder, M.D.: The protection of information in computer systems. Proc. IEEE 63, 1278–1308 (1975)
11. Tuch, H.: Formal Memory Models for Verifying C Systems Code. PhD thesis, School Comp. Sci. & Engin., University NSW, Sydney 2052, Australia (August 2008)
12. Tuch, H., Klein, G., Norrish, M.: Types, bytes, and separation logic. In: Hofmann, M., Felleisen, M. (eds.) 34th POPL, Nice, France, pp. 97–108 (January 2007)
13. Winwood, S., Klein, G., Sewell, T., Andronick, J., Cock, D., Norrish, M.: Mind the gap: A verification framework for low-level C. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 500–515. Springer, Heidelberg (2009)

Static Contract Checking with Abstract Interpretation

Manuel Fähndrich and Francesco Logozzo

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

Abstract. We present an overview of Clousot, our current tool to statically check CodeContracts. CodeContracts enable a compiler and language-independent specification of Contracts (precondition, postconditions and object invariants).

Clousot checks every method in isolation using an assume/guarantee reasoning: For each method under analysis Clousot assumes its precondition and asserts the postcondition. For each invoked method, Clousot asserts its precondition and assumes the postcondition. Clousot also checks the absence of common runtime errors, such as null-pointer errors, buffer or array overruns, divisions by zero, as well as less common ones such as checked integer overflows or floating point precision mismatches in comparisons. At the core of Clousot there is an abstract interpretation engine which infers program facts. Facts are used to discharge the assertions. The use of abstract interpretation (*vs* usual weakest precondition-based checkers) has two main advantages: (i) the checker automatically infers loop invariants letting the user focus only on boundary specifications; (ii) the checker is deterministic in its behavior (which abstractly mimics the flow of the program) and it can be tuned for precision and cost. Clousot embodies other techniques, such as iterative domain refinement, goal-directed backward propagation, precondition and postcondition inference, and message prioritization.

1 Introduction

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

The CodeContracts project [16] at Microsoft Research aims at bridging the gap between practice and formal specification and verification using the principle of least interference in the programmer's existing workflow. The main insight of CodeContracts is that program specifications can be authored as code [17]. Contracts take the 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: the 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, type-checking . . .), all the IDE support (intellisense, code refactoring . . .) 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 tools to automatically generate the documentation (ccdoc), to perform runtime checking (ccrewrite) and to perform static checking (cccheck, internally called Clousot). The tools are available for download at

$$\texttt{http://msdn.microsoft.com/es-ar/devlabs/dd491992(en-us).aspx}$$

A main difference of our static contract checker, with respect to similar and existing ones is that it is based on abstract interpretation [9] instead of solely relying on a theorem prover (automatic [20,2,19] or semiautomatic [3]). The use of abstract interpretation allows the checker to focus on some properties of interest, as for instance non-nullness, linear arithmetic or array invariants while forgetting more complex or unusual ones such as existentially quantified or arbitrarily universally quantified properties. An abstract interpretation-based static checker has the advantage of being more automatic and tunable than theorem prover-based ones. For instance, it can automatically compute loop invariants, which frees the programmer from the burden of specifying (often self-evident) loop invariants. The built-in abstract domains are optimized for the properties of interest, so that the precision/cost ratio can be finely set. Furthermore, the analysis is deterministic, in that it does not depend on internals of theorem provers such as random seeding, quantifier instantiation, or matching loops.

A number of automated verification tools based on separation logic essentially also use a fixpoint computation based on abstract interpretation [23,40,14]. Our approach is very similar to theirs in that our optimistic heap abstraction (Section 4) effectively uses an implicit form of separation logic, where all heap locations are assumed to be joined by separating conjunctions. The optimistic aspect of our heap abstraction arises in the fact that we don't ever try to prove the separation itself, we only assume it. This approach works well for program parts that do not depend on complicated aliasing or data structure invariants. Therefore our approach is not suited for proving properties about pointer relations themselves.

2 CodeContracts by Example

The class in Fig. 1 in an example of a C# class annotated with CodeContracts specifications. Contracts are defined by means of calls to static methods of a `Contract` class, part of .Net since $v4.0$. The class implements a simple stack of non-null objects. Externally, one can create a stack, can push or pop elements, can inquire about the number of stack elements and whether the stack is empty or not. Internally, the stack is backed-up by two fields: a growing array of objects containing the stack elements and a pointer to the next free position in the stack.

```
public class NonNullStack<T> where T : class
 {
   private T[] arr;
   private int nextFree;

   [ContractInvariantMethod] /* Define the object invariant */
   void ObjectInvariant()
   {
     Contract.Invariant(arr !=null);
     Contract.Invariant(nextFree >= 0);
     Contract.Invariant(nextFree <= arr.Length);
     Contract.Invariant(Contract.ForAll(0, nextFree, i => arr[i] != null));
   }

   public NonNullStack(int len)
   {
     Contract.Requires(len >= 0); /* Method precondition */

     this.arr = new T[len];
     this.nextFree = 0;
   }

   public void Push(T x)
   {
     Contract.Requires(x != null);

     if (nextFree == arr.Length)
     {
       var newArr = new T[arr.Length * 2]; /* bug here */
       for (int i = 0; i < nextFree; i++) newArr[i] = arr[i];
       arr = newArr;
     }
     this.arr[nextFree++] = x;
   }

   public T Pop()
   {
     Contract.Requires(!this.IsEmpty);
     Contract.Ensures(Contract.Result<T>() != null); /* Method postcondition */

     return this.arr[--nextFree];
   }

   public bool IsEmpty { get { return this.nextFree == 0; } }
   public int Count { get { return this.nextFree; } }
 }
```

Fig. 1. A (buggy) implementation of a stack of non-null values annotated with CodeContracts. `Contract.Requires` specifies the precondition, `Contract.Ensures` specify the postcondition, `Contract.Result` denotes the return value (not expressible in C#). The attribute `ContractInvariantMethod` tags the method containing the object invariant (specified with the `Contract.Invariant`).

As a programmer, one would like to express some simple properties about those fields. The first property is that the array is never null and that the pointer can never be negative. Furthermore, the stack pointer can never be larger than the array length (it can be equal when the stack is full). Finally, all the elements in the interval $\mathtt{a}[0] \ldots \mathtt{a}[\mathtt{nextFree} - 1]$ should be not-null.

2.1 Specification

The formal specification with CodeContracts of those invariants is given by the method `ObjectInvariant` if Fig. 1. CodeContracts require the object invariant to be specified in a void method annotated with the attribute `ContractInvariantMethod`. The object invariant method body can only contain calls to `Contract.Invariant`, which specify the object invariant. Valid conditions for contracts are language expressions, including those containing method calls (provided the callee is marked with the [`Pure`] attribute) augmented with dummy methods to specify limited universal (`Contract.ForAll`) and existential quantification (`Contract.Exists`).

Preconditions are expressed via `Contract.Requires`. In the example, the precondition of the `NonNullStack` constructor requires the caller to pass a non-negative initial size for the stack.

Postconditions are expressed via `Contract.Ensures`. In the example, the postcondition of the method `Pop` ensures that the returned value is not-null. The void method call `Contract.Result` $\langle$`T`$\rangle$() is used to denote the return value of the method, which is not directly expressible in the source language.

CodeContracts, being simple method calls, are totally transparent to the compiler, and thanks to the shared type system in .Net, also to the different languages. Programmers can author Contracts in their favorite .Net language (C#, VB, F# ...). The compiler compiles contracts to straight CIL (Common Intermediate Language [15]). Our tool extracts the contracts from the CIL and use them for multiple purposes: Documentation generation, Runtime checking and Static checking (Clousot).

2.2 Static Checking

Clousot analyzes every method in isolation, using the usual assume/guarantee reasoning. 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.

From a user-perspective, Clousot makes the distinction between explicit and implicit assertions (or proof obligations). *Explicit* proof obligations are those provided by the user as specifications or as an explicit assertion. In the running example, the object invariant and the postcondition of `Pop` are the assertions to be proved. *Implicit* proof obligations are those defined by the CIL language semantics, to avoid runtime errors such as null deference, index out of range for arrays or overflows for checked arithmetic expressions, but also buffer overruns which do not cause an exception to be thrown, but may compromise the stability (and security) of the program. In the default configuration, Clousot only checks the explicit proof obligations, to avoid overwhelming the user with too many warning messages. At first, we want the user to focus on boundary specifications. Once those are resolved (possibly going to zero warnings), the programmer can (selectively) enable the checking of the implicit proof obligations.

The analysis proceeds by performing some abstract interpretations of the method body, where all the contracts are turned into asserts or assumes. Clousot contains abstract domains tailored to specific properties of interest, such as heap location equalities, non-nullness, linear arithmetic, disjunctions and simple universally quantified facts. Those properties are enough to analyze and verify the example of Fig. 1.

Static Contract Checking. To prove the object invariant for `NonNullStack`, one must be able to track nonnullness (to prove that $\texttt{arr}! = \texttt{null}$), linear arithmetic relationships (to prove that $\texttt{0} \leq \texttt{nextFree} \leq \texttt{arr.Length}$) and quantified facts (to prove that $\forall i \in [0, \texttt{arr.Length}).\texttt{arr}[i]! = \texttt{null}$). The most interesting case is the implementation of `Push`. First it checks if the backing array is full. If it is, it allocates an array twice as large and copies all the original elements into it. Finally it updates the array with `x` and increments the stack pointer.

The nonnull analysis infers that in both if-branches $\texttt{arr} != \texttt{null}$, so it concludes that the first conjunct of the invariant is satisfied.

The numerical analysis infers in one case (array full) that $\texttt{0} \leq \texttt{nextFree} \leq \texttt{arr.Length}$ and in the other that $\texttt{0} \leq \texttt{nextFree} < \texttt{arr.Length}$, so that $\texttt{0} \leq \texttt{nextFree} \leq \texttt{arr.Length}$ holds before the array store. The method exit point is reached only if the store was successful, *i.e.*, the index was inbounds, so that the abstract element can be refined to $\texttt{0} \leq \texttt{nextFree} < \texttt{arr.Length}$, and hence prove the other two conjuncts of the object invariant.

The universally quantified component of the object invariant is a little bit trickier. We know that the elements $\texttt{arr}[0] \ldots \texttt{arr}[\texttt{nextFree} - 1]$ are not null (from the object invariant), and that the element to be pushed is not null (from the precondition). When there is still space, we can easily conclude that the elements $\texttt{arr}[0] \ldots \texttt{arr}[\texttt{nextFree} - 1], \texttt{arr}[\texttt{nextFree}]$ are all not null. When there is no more space, a new array is allocated and all the elements are copied into it. Proving that $\texttt{newArr}[0] \ldots \texttt{newArr}[\texttt{nextFree} - 1]$ are all not null requires inferring the quantified loop invariant $\forall j \in [0, \texttt{i}].\ \texttt{arr}[j]! = \texttt{null}$. In Clousot we have new abstract domains to infer such invariants efficiently (Sect. 5.4).

Static Runtime-Error Checking. Once all the boundary contracts are proved, the user can opt-in to prove the absence of common runtime errors in the implementations. For instance, the user can turn on the non-null and array bounds checking. Then every time a field, an array, and in general a reference is accessed, Clousot will try to prove that such a reference is not null. In our example, Clousot will prove the absence of null references in the class. As for array bounds checking, every time an array is created, read or written, Clousot will try to prove that the access is in-bounds. For instance for an array store $\texttt{a}[\texttt{exp}]$

Clousot will emit the condition $0 \leq$ `exp` (underflow) and `exp` $<$ `a.Length` (overflow). In our example the most interesting case is `Push`. When the stack is full, then a new array is allocated and all the elements are copied into it. To prove the array accesses correct, Clousot infers the loop invariant `0` $\leq$ `i` $\leq$ `nextFree`, which combined with the guard `nextFree` $==$ `arr.Length`, the array creation postcondition `newArr.Length` $=$ `2` $*$ `arr.Length` and the loop guard, allows proving the safety of the `newArr` store and `arr` read inside the loop. At the end of the loop, one only knows that `0` $\leq$ `nextFree` $\leq$ `newArr.Length`, which is not enough to prove the safety of the next store instruction. In fact, when `a.Length` $= 0$, then $0 =$ `nextFree` $=$ `newArr.Length` and the store is indeed causing an overrun. The programmer can fix it by changing the allocation expression to `arr.Length` $*$ `2` $+$ `1`, in which case Clousot will discover that `nextFree` $<$ `newArr.Length`, and hence validating the store.

The programmer can be more picky, and may want to prove more things about the program. He/she can turn on the arithmetic obligations switch in Clousot to check for common arithmetic errors such as division by zero or the overflow of checked expressions. In the particular example Clousot discovers that the array allocation `newint[arr.Length` $*$ `2` $+$ `1]` may cause an overflow exception. The expression `arr.Length` $*$ `2` $+$ `1` may overflow to a negative `Int32`, that when converted into a `UInt32` will cause an overflow. Inserting an explicit check against overflow will remove the warning.

Finally, Clousot helps to reduce the annotation burden by inferring some "easy" postconditions. In the default settings, Clousot infers postconditions only for: (i) properties and (ii) methods that return a non-null value. For the getter `IsEmpty` in our example, Clousot infers the postcondition `Contract.Result` $\langle$`bool`$\rangle$`()` $==$ (`this.nextFree` $==$ `0`). The postcondition is then propagated to all the call sites, so that for instance one can prove the safety of the array load in the `Pop` method.

3 The Analysis

Target Language. Clousot works at the bytecode level (CIL, Common Intermediate Language [15]). This is different from many other static analyzers, which work at the source level. There are several advantages of working at the bytecode level. First, the analysis is language independent: Clousot can analyze code produced by any compiler generating CIL (C#, VB, F# . . .). Second, the analysis leverages the compiler to give semantics to complex constructs. For instance C# 3.0 introduced type inference for locals. The type inference algorithm is quite complicated, but once the compiler inferred all the types, then it generates straight IL. A source level analyzer for C# 3.0 would have to replicate the compiler type inference algorithm. A bytecode level analyzer can simply analyze the compiled IL. Third, the analysis is stable among different versions of the same language: languages change, CIL stays the same. For instance, C# 4.0 added many features over C# 3.0, such as the `dynamic` keyword or named parameters. A source level analyzer would have required (at least) a new parser to adapt to the new syntax. To the bytecode level analyzer the upgrade is totally transparent. Fourth, Contracts (serialized and persisted as CIL) do not need to be decompiled to some high level description.

Bytecode analysis has drawbacks too [31]. The main one is that high-level structure is lost, so that some additional analysis must be carried out to recover some of the information. Furthermore classical static analysis refinement techniques such as loop unrolling are harder to implement.

Phases. Clousot has three main phases: Inference, Checking and Propagation. During the inference phase, the methods of the assemblies to analyze are sorted, so that callees are analyzed before their callers when possible. If there is a cyclic dependency between methods, it is broken by picking one method in the chain. For each method under analysis, its IL is read from the disk and its contracts are extracted. Then the method is analyzed. By analysis we mean a fixpoint computation with widening over a suitable abstract domain. First, aliasing is resolved (under some optimistic hypotheses) and the method code is abstracted into a *scalar* program. Then further analyses are run on the top of it to infer facts on the program. In the checking phase, the (explicit and implicit) proof obligations are collected, and the inferred facts are used to discharge them. If a proof obligation cannot be discharged, then the analysis is refined. If the more refined analysis fails, then a warning is reported to the user. Eventually, the inferred facts are used to materialize method postconditions that are attached to the method under analysis, and hence automatically propagated to the call sites.

4 Basic Framework

The inference phase is in its turn divided into two phases: (i) the scalar program construction and expression recovery; and (ii) the fact discovery. The first phase takes care of building the control flow graph (CFG), extracting the contracts and inserting them at the right spots, get rid of the stack, perform a heap analysis, and reconstruct larger expressions lost during compilation. The output of this phase is a program in scalar form. The second phase takes as input the scalar program, and performs a series of value analyses to infer facts for each program point in the method body.

Contract Extraction and CFG Construction. The code to be analyzed is factored into subroutines: one subroutine per method body, one subroutine for a method's preconditions, and one subroutine for a method's postconditions. The actual code to be analyzed is then formed by inserting calls to appropriate contract subroutines in the method body. Additionally, at each method call-site, we insert a call to the precondition subroutine of the called method just prior to the actual call, and a call to the corresponding postcondition subroutine immediately following the call. The actual contract calls to `Contract.Requires` or `Contract.Ensures` turn into either `assert` or `assume` statements depending on their context. `Requires` on entry of a method turn into `assume` and `Ensures` on exit of a method turn into `assert`. Conversely, at call-sites, `Requires` turn into `assert`, and `Ensures` turn into `assume`. Conditional branches are expanded into non-deterministic branches with `assume` statements on the outgoing edges. In this manner, all conditions are simply sequences of CIL instructions, no different than ordinary method body code, and all assumptions are assume statements, and all explicit proof-obligations are assert statements.

Heap Abstraction. The heap is abstracted by a graph, the Heap-graph, which maintains equalities between access paths (rooted in a local or a method parameter). Nodes in the graph denote symbolic values or heap locations, and edges denote containment or field selection. The intuitive meaning is that if two paths in the graph lead to the same node, then: (i) in the concrete executions they *always* represent the same value; and (ii) this value is *symbolically* denoted by the same symbolic value `sv`. The heap graph abstraction is optimistic in that it makes certain assumptions about non-aliasing of data structures that may not be correct in all executions. It is the only place in Clousot where such assumptions are made. Namely we assume that memory locations not explicitly aliased by the code under analysis are non-aliasing. This is clearly an optimistic assumption, but works very well in practice. Second, we guess the set of heap locations that are modified at call-sites (we do not require programmers to write heap modification clauses). Our guesses are often conservative, but may be optimistic if our non-aliasing assumptions are wrong. These assumptions allow us to compute a value numbering for all values accessed by the code, including heap accessing expressions. We also introduce names for uninterpreted functions marked as [`Pure`] by the programmer. This provides reasoning over abstract predicates. Finally, abstracting the heap also removes old-expressions in postconditions that refer to the state of an expression at the beginning of the method.

To compute the value numbering, we break the control flow of the analyzed code into maximal tree fragments. The root of each tree fragment is a join point (or the method entry point) and is connected by edges to predecessor leafs of other tree fragments. The set of names used by the value numbering is unique in each tree fragment. Edges connecting tree leafs to tree roots contain a set of assignments effectively rebinding value names from one fragment to the names of the next. The resulting code is in mostly passive form, where each instruction simply relates a set of value names. The assignments on rebinding edges between tree fragments provide a way to transform abstract domain knowledge prior to the join from one set of value names to the next, so that the join can operate on a common set of value names. The rebinding acts as a generalization of ϕ-nodes. In contrast to ϕ-nodes which provide a join for each value separately, our rebindings form a join for the entire state simultaneously, which is crucial to maintain relational properties.

Example 1. Consider the code snippet in Fig. Fig. 2. The heap analysis captures the fact that `p.b` and `a.b` are aliases starting from program point $(*)$.

```
void HeapExample(bool b, A a, P p)
{
  p.b = a.b; // (*)

  if (b)
    a.b.x = 12;
  else
    p.b.x = 4;

  Contract.Assert(a.b.x >= 4);
}
```

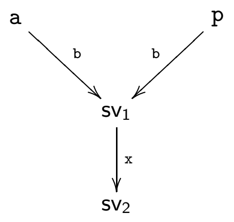


Fig. 2. A simple program and the corresponding Heap abstraction

The heap graph looks like the one in Fig. 2 (intermediate address nodes for locals and fields have been omitted for brevity) where symbols on edges denote the fields being selected, and sv_1 is the symbolic value of a.b, and sv_2 is the symbolic value of a.b.x. □

In the following we let sv(p) denote the symbolic value assigned by the heap analysis to the path p.

Expression Reconstruction. The expression reconstruction analysis allows to recover some of the structure of Boolean and Arithmetic expressions that may have been lost during the compilation. The analysis is similar in many aspects to the symbolic abstract domain of [36]. A main difference is that the depth of exploration for the expression reconstruction is dynamically chosen by the particular analysis (essentially performing a widening). A comprehensive discussion of the pros and the cons of a bytecode level analysis is in [31].

5 Fact Inference

5.1 NonNull Analysis

The NonNull analysis discovers those references which are definitely not-null or definitely null. Given a reference r, the analysis assigns r a value in the flat lattice $\perp \sqsubseteq \mathtt{N}, \mathtt{NN} \sqsubseteq \top$, with N meaning that the reference is *always* `null` and NN meaning that the reference is *never* `null`.

5.2 Numerical Analysis

The numerical analysis discovers ranges and linear arithmetic relationships between symbolic values. Those relationships are then used to discharge proof obligations containing numerical conditions. The numerical analysis is a usual forward fixpoint computation with widening [7] parametrized by a numerical abstract domain.

Transfer functions corresponding to CIL instructions are parametrized by the underlying abstract domain. For instance, when an array store `ldelem` a[exp] is encountered, two numerical constraints are pushed to the numerical abstract domain: $0 \leq \mathsf{sv}(\mathsf{exp})$ and $\mathsf{sv}(\mathsf{exp}) < \mathsf{sv}(\mathsf{a.Length})$.

Example 2. Let us consider the example in Fig. 2. A simple numerical domain infers that $\mathsf{sv}_2 = 12$ at the end of the *true* branch of the conditional, and $\mathsf{sv}_2 = 4$ at the end of the *false* branch. As a consequence, at the exit point of the conditional $4 \leq \mathsf{sv}_2 \leq 12$, which is sufficient to prove the assertion. □

Thresholds are used to improve the precision of the widening (as in [4]). The thresholds are collected from the constants appearing in assumptions and assertions in the method. The numerical analysis *assumes* the common case that arithmetic expressions do not overflow, but it explicitly *checks* it in presence of checked operations [1]. Therefore our assumption can be easily checked by instructing the compiler to threat all the operations as checked. Clousot will then try to prove that they do not overflow.

Numerical Abstract Domains. They abstract the values of numerical program variables. In the literature many numerical abstract domains have been developed with different precision/cost tradeoffs. Intervals [9] infer properties in the form $\mathsf{x} \in [a, b]$, where $a, b \in \mathbb{Z} \cup \{-\infty, +\infty\}$. Intervals are very efficient yet unsuitable for symbolic reasoning as they do not keep track of relations among different variables. At the opposite end of the precision spectrum Polyhedra [13] capture arbitrary linear inequalities in the form of $\sum a_i \cdot \mathsf{x}_i \leq b$. Polyhedra are very precise yet expensive (the worst case, easily attained in practice is exponential). In between these two domains, other domains (*weakly* relational) have been developed to tune the precision/cost ratio. Examples include Octagons [35] ($\pm\mathsf{x} \pm \mathsf{y} \leq b$), TVPI [39] ($a_1\mathsf{x} + a_2\mathsf{y} \leq b$) or Octahedra [6] ($\sum \pm\mathsf{x}_i \leq b$). In Clousot, we first tried using some of these domains, but we found them unfit for our purposes. For instance, Octagons introduce a non-negligible slowdown (the complexity is cubic in the number of variables, with a large multiplicative constant). A known technique to have Octagons scale up is bucketing (or packing), where buckets are restricted to a certain fixed number, and some weak relations are kept by using pivot variables. We rejected buckets, as they make the analysis result dependent on the order in which the heap analysis generates the variables, introducing a degree of non-determinism in our analysis which we prefer to avoid. We also tried Polyhedra, but early results turned out to be very bad [18]. As a consequence we developed a series of new numerical abstract domains, refining and combining existing ones. They are mainly validated by empirical experimenting and tuning.

DisIntervals. DisIntervals are a simple extensions of Intervals to a finite disjunction. Formally they are an abstraction of the disjunctive completion of Intervals [8]. Elements of Disintervals are normalized sequences of non-overlapping intervals: $[a_0, b_0], \dots [a_i, b_i], [a_{i+1}, b_{i+1}] \dots [a_n, b_n]$ with the property that only a_0 can be $-\infty$; only b_n can be $+\infty$ and that $\forall i \in [0, n-1]. b_i < a_{i+1}$. Usual

[1] The CIL instruction set has *checked* counterparts for all the arithmetic operations which cause an exception to be thrown if an overflow has occurred.

operations on Intervals can be easily lifted to Disintervals (only the widening needs some care). DisIntervals present a very cheap way to represent non-relational disjunction as well as common "negative" information. For instance $\mathsf{x} \in [-\infty, -1], [1, 5], [50, +\infty]$ is a compact representation for $\mathsf{x} \neq 0 \wedge \mathsf{x} \neq 6 \wedge \ldots \mathsf{x} \neq 49$. This kind of information is needed for instance when dealing with enumerations.

In early versions of Clousot we had one abstract domain for Intervals and one for simple disequalities. It turned out that combining the two into the Disinterval abstract domain improves the precision, simplifies the implementation, and produces no observable slow-down in our tests and experiments.

Zones. DisIntervals, or Intervals are non-relational domains which are useful in many situations. However, in modular static analysis one needs to perform some form of symbolic reasoning. The easiest one involves simple upper bounds.

Example 3. Let us consider the method `AllZero` in Fig. 3. (Dis)Intervals infer the loop invariant $\mathsf{sv(i)} \in [0, +\infty]$, which is enough to prove that the array store will not cause an underflow. To prove no overflow will ever occur, one needs to propagate the constraint $\mathsf{sv(i)} < \mathsf{sv(a.Length)}$. To prove the assertion at the end of the loop, one needs to infer the loop invariant $\mathsf{sv(i)} \leq \mathsf{sv(a.Length)}$, which together with the loop exit condition is exactly the assertion. □

```
void AllZero(int[] a)
 {
  Contract.Requires(a != null);
  int i;
  for(i = 0; i < a.Length; i++) a[i] = 0;
  Contract.Assert(i == a.Length);
}
```

Fig. 3. Example requiring a numerical abstract domain able to perform symbolical reasoning

In Clousot, WeakUpperBounds capture strict upper-bounds $\mathsf{x} < \mathsf{y}_0, \ldots \mathsf{y}_i$ and WeakUpperBoundsEqual capture upper-bounds $\mathsf{x} \leq \mathsf{y}_0, \ldots \mathsf{y}_i$. They enable very efficient implementations in terms of maps. We call Disintervals combined with WeakUpperBounds and WeakUpperBoundsEqual Pentagons [32]. Pentagons are essentially a weak form of the zones abstract domains [33]. The major difference is that Pentagons avoid performing the costly closure operation, relying instead on *hint* operators to keep acceptable precision at join points [29].

Linear Equalities. We use the abstract domain of linear equalities [26] to infer and propagate relations in the form $\sum a_i \cdot \mathsf{x}_i = b$. The linear equalities domain enables a very efficient implementation in terms of sparse arrays which largely compensates for the cubic cost. When combined with Pentagons, Linear Equalities can produce very powerful analyses at a moderate cost.

Example 4. Let us consider the example in Fig. 4 (taken from [38]): At loop exit, (Dis)intervals infer $\mathsf{sv(i)} \in [1, +\infty], \mathsf{sv(j)} \in [-\infty, +\infty], \mathsf{sv(x)} \in [0, 0], \mathsf{sv(y)} \in [-\infty, +\infty]$ and Linear Equalities infer $\mathsf{sv(x)} - \mathsf{sv(y)} = \mathsf{sv(i)} - \mathsf{sv(j)}$. At the assertion we can then conclude that $\mathsf{sv(i)} = \mathsf{sv(j)}$. □

```
void Foo(int i, int j)
{
  var x = i, y = j;

  if(x <= 0) return;
  while(x > 0)
  { x--; y--; }

  if(y == 0)
   Contract.Assert(i == j);
}
```

Fig. 4. Example needing the inference of the loop invariant $\langle \mathtt{x} - \mathtt{y} = \mathtt{i} - \mathtt{j}, \mathtt{x} \in [0, +\infty]\rangle$, easily obtained by combining Linear Equalities and Intervals

Please note that even if the assertion has a shape that would fit other weak relational domains, proving it require inferring a relation involving four variables, which is out of reach of those domains. This an extremely common case that we found over and over.

Combination of Domains. Every single abstract domain sketched above is weak by itself, but their combination can produce very powerful analyses [10]. The basis of the combination of numerical abstract domains in Clousot is the *reduced* product [9]. Given two abstract domains A_1 and A_2, the *cartesian* product $\mathsf{A}_1 \times \mathsf{A}_2$ is equivalent to running the two analyses separately, so that no precision gain is obtained by the composition (worse, in general it can slow down the analysis). If the two domains are allowed to communicate, by either pulling or pushing information, then the analysis precision can be dramatically improved. The example of the previous section is an example of pushing: By *pushing* the information that $\mathsf{sv}(\mathsf{x}) = 0$ at the end of the loop, the abstract state for linear equalities is refined to $\mathsf{sv}(\mathsf{x}) - \mathsf{sv}(\mathsf{y}) = \mathsf{sv}(\mathsf{i}) - \mathsf{sv}(\mathsf{j}) \wedge \mathsf{sv}(\mathsf{x}) = 0$. Please note that linear equalities alone cannot infer that $\mathsf{sv}(\mathsf{x}) = 0$, as this is a consequence of the loop invariant $\mathsf{sv}(\mathsf{x}) \geq 0$, which is not a linear equality. Pulling is mainly used during the fixpoint computation when transfer functions may *explictly* ask other domains to refine some information, or if some relation holds. For instance suppose that we have to evaluate the expression $\mathsf{sv}(\mathsf{u}) - \mathsf{sv}(\mathsf{w})$ in an interval environment where $\mathsf{sv}(\mathsf{u}) \in [0, +\infty], \mathsf{sv}(\mathsf{w}) \in [0, +\infty]$. With no additional information the result can be any `Int32`. Intervals can *pull* information from other domains (*oracles*), for instance asking if $\mathsf{sv}(\mathsf{w}) < \mathsf{sv}(\mathsf{u})$. The oracle can return four possible outcomes: $\top$, meaning *"I do not know"*; $\bot$ meaning this program point is unreachable, so the evaluation simply returns $\bot$; true so that the result can be refined to $[1, +\infty]$; false meaning that $\mathsf{sv}(\mathsf{w}) \geq \mathsf{sv}(\mathsf{u})$ holds, so that the result can be refined to $[-\infty, 0]$. To avoid computing a fixpoint computation among the different abstract domains at every single step of the analysis, the domains are ordered according to a tree structure (as in [10]) where the most precise yet expensive domains are at the root, and the less precise yet cheaper are towards the leafs. Every domain is allowed to pull information from every domain, but only higher-rank domains can push information to lower-rank ones.

Subpolyhedra. In the general setting of contract checking, arbitrary linear inequalities are needed for effective symbolic reasoning. For instance in the example in Fig. 5, one needs to infer the loop invariant $0 \leq \mathsf{sv}(\mathsf{i}) + \mathsf{sv}(\mathsf{index}) \leq \mathsf{sv}(\mathsf{output.Length})$.

Using the classical Polyhedra turned out to be far too expensive [18]. We are aware that many advances have been made to optimize them [1,24], but we are

```
void ArrayCopy(int[] input, int[] output, int index)
{
  Contract.Requires(index >= 0);
  Contract.Requires(output.Length - index >= input.Length);

  for (var i = 0; i < input.Length; i++) output[i+index] = input[i];
}
```

Fig. 5. Simple example where fully fledged relational numerical domains are needed

still skeptical that they can scale up to the needs of Clousot's customers. Classical Polyhedra have a double representation for an abstract state: geometrical (where the Polyhedra is expressed as a set of points and generators) and algebraic (maintaining the tableau of equations defining the polyhedron). Some abstract operations are very efficient in one form, some in another. Converting from one form to its dual is very expensive (exponential) and it has been shown that it cannot be done faster [27]. Hence we developed a new abstract domain, Supolyhedra, which is as *expressive* as Polyhedra, but which gives away some of the inference power. The main, simple idea, is to split a linear inequality $\sum a_i \cdot \mathsf{x}_i \leq b$ into an equality and an interval via a slack variable β: $\sum a_i \cdot \mathsf{x}_i = \beta \wedge \beta \in [-\infty, b]$. Each of the two conjuncts is handled by a separate abstract domain, i.e., linear equalities and intervals. There are two main challenges here. The first one is to have a precise enough join, the pairwise join being simply to rough. The second one is to have an effective reduction algorithm to get the tightest bounds on the intervals. We have defined in [29] a join (and widening) operator which allow fine tuning the two points above, *de facto* defining a family of abstract domains, where the precision/cost ratio can be adjusted: more precise domains are obtained by improving the *hints* [28] at join/widening points and the reduction subroutine. In our tests Subpolyhedra scales to hundreds of variables, going well beyond the current state of the art of Polyhedra implementations.

5.3 Floating Point Values

We have an implementation of Intervals supporting the IEEE 754 standard. We have not yet extended this support to relational domains, as for instance [34,5], so that the amount of reasoning that can be done on floats is very limited. We have an analysis to figure out possible precision mismatches in double comparisons caused by implicit conversions between 80 and 64 bits of precision. Such conversions may introduce subtle bugs. This is best illustrated by the example in Fig. 6.

One may expect the postcondition to trivially hold. However, using an automatic test generation tool as *e.g.* PEX [41] one can easily find counterexamples to the postconditions! The ECMA standard [15] allows locals (including parameters) to be passed with the full precision of the architecture, whereas fields should *always* be truncated to 64 bit doubles. In an x86 architecture, double registers are 80 bits long. As a consequence, `amount` is passed as an 80 bit value, the result of `this.balance` + `amount` is stored in a CPU register

```
private double balance;
public void Deposit(double amount)
{
  Contract.Requires(amount >= 0.0);
  Contract.Ensures(this.balance == Contract.OldValue(balance) + amount);

  balance = balance + amount;
}
```

Fig. 6. Example showing problems induced by the extra-precision for `double`s allowed by the ECMA standard. The field `balance` is stored into a 64 bits memory location whereas the result of `balance` + `amount` is stored into a 80 register.

(80 bits), but when written back to memory, it gets truncated to 64 bits. As a consequence the postcondition may be violated at runtime for specific values of `amount`. Clousot tracks floating point types of a symbolic values according to the flat lattice $\bot \sqsubseteq$ `Float`, `CPUFloat` $\sqsubseteq \top$, `Float` $\neq$ `CPUFloat`. In the example, Clousot infers `balance` + `amount` : `CPUFloat` and `balance` : `Float`, and hence issues a warning for a possible precision mismatch. An explicit cast forces the truncation: the correct postcondition is hence `balance` == (`double`) (`Contract.OldValue`(`balance`)+`amount`).

5.4 Arrays and Collections

The abstract domains for scalar values are lifted to sequences (like arrays or collections) via a parametric segmentation functor [12]. The functor automatically and semantically divides (*e.g.*) arrays into sequences of consecutive non-overlapping possibly empty segments. Segments are delimited by sets of boundary expressions and abstracted uniformly. The overhead of the analysis is very low (around 1% on large framework libraries). Once again we developed a new (functor) abstract domain as existing solutions turned out either to require too much extra-assistance from the user [25] or to be inherently not-scalable [21,22].

Example 5. At the end of the `for` loop of the (incorrect version of the) method `Push`, the array analysis associates the following two abstract elements to the arrays:
$$\begin{array}{rl} \texttt{arr} & \mapsto \{\texttt{0}\}\texttt{NN}\{\mathsf{sv(i)}, \mathsf{sv(nextFree)}, \mathsf{sv(arrLen)}\}? \\ \texttt{newArr} & \mapsto \{\texttt{0}\}\texttt{NN}\{\mathsf{sv(i)}, \mathsf{sv(nextFree)}\}?\texttt{N}\{\mathsf{sv(newArrLen)}?\} \end{array}$$

stating that all the elements of `newArr` up to `nextFree` are not-null, but also that $\mathsf{sv(i)} = \mathsf{sv(nextFree)}$ (expressions in bounds are equal) and that it may be the case $0 = \mathsf{sv(nextFree)} = \mathsf{sv(newArrLen)}$, in which case the `newArr` is empty (? denotes the fact that successive segments may be equal). □

Example 6. For the method `AllZero` of Fig. 3, at loop exit the analysis discover the invariant $\mathsf{a} \mapsto \{\texttt{0}\}[\texttt{0}, \texttt{0}]\{\mathsf{sv(i)}, \mathsf{sv(a.Length)}\}?$ which compactly represents $\forall j \in [0, \mathsf{a.Length}).\mathsf{a}[j] = 0 \wedge \mathsf{sv(i)} = \mathsf{sv(a.Length)} \wedge 0 \leq \mathsf{sv(i)}$.

6 Checking

Assertion Crawling. The code of the method under analysis is crawled to collect a set of proof obligations $\mathcal{P}$. Proof obligations are either explicit or implicit. Explicit proof obligations are either: (i) preconditions at call sites; (ii) explicit assertions; or (iii) postcondition for the current method. Checking of explicit proof obligations is always on. Implicit proof obligations are induced by the CIL semantics. For a reference access r, a non null proof obligation $r \neq \mathtt{null}$ is emitted. For an array creation with size `exp`, a proof obligation $0 \leq \mathtt{exp}$ is emitted. For an array load or store with index `exp`, the two proof obligations $0 \leq \mathtt{exp}$ and $\mathtt{exp} < \mathtt{svLen}$ are emitted. Similarly for buffer accesses, divisions, negation of minint, overflow checking and floating type mismatches. The checks for implicit proof obligations (such as non-null dereferencing and array bound checks) can be individually activated by the user. The rationale is to avoid drowning the user with too many warnings and instead have him/her first focus on the contracts.

Direct Checking. For each proof obligation $\langle \mathtt{pc}, c\rangle \in \mathcal{P}$ Clousot individually asks each of the analyses if at program point `pc` the condition c holds. Each analysis implements a specialized decision procedure (in the numerical and the array analysis those specialized decision procedures are also invoked during the fixpoint computation to refine the analysis itself). The analysis fetches the abstract state at program point `pc`, and checks if it implies c. Fetching may cause a re-run of a part of the analysis, as for performance and memory considerations we only save abstract states at some specific program points (*e.g.* loop heads as in [4]). There are four possible check outcomes: true, meaning that c holds for all the possible executions *reaching* `pc`; false, meaning that there is no execution reaching `pc` such that c holds; $\perp$, meaning that the program point `pc` is unreached (dead code); and $\top$ meaning that the analysis does not have a definite answer. Direct checking $\langle \mathtt{pc}, c\rangle$ is aborted as soon as an outcome different from $\top$ is reported. This approach may fail to report the most precise answer, produced by the meet of all the analyses outcomes. We do so mainly for performance reasons (projects typically contain tenths of thousands of proof obligations to discharge).

Domain Refinement. If all the analyses had $\top$ as outcome, then Clousot refines the analysis. One first way of refining the analysis is to re-analyze the method body with a more precise abstract domain. Clousot implements an iterative strategy in which first less precise abstract domains are used (*e.g.* the numerical analysis instantiated with Pentagons) then moving to more precise yet expensive domains. In the worst case, one may always resort to the most expensive domain (*e.g* Subpolyhedra with all the hints on and the Simplex-based reduction [29]). Empirically we noticed that refinement pays off since the number of cases where one needs the most expensive domains is relatively small.

```
string Nums(int a)
{
  Contract.Requires(a > 0);

  string s = null;
  var i = 0
  /* 1 */
  for (; i < a; i++) { s += i.ToString(); /* 2 */}

  /* 3 */
  Contract.Assert(s != null);

  return s;
}
```

Fig. 7. Example showing the combination of analyses via backward goal propagation. The NonNull analysis discovers that $\texttt{s}! = \texttt{null}$ at 2, and the Numerical analysis discovers that the path $1 \rightarrow 3$ is unfeasible.

Goal Directed Backwards Analysis. If domain refinement is not good enough to discharge a proof obligation, we propagate the condition backwards. Essentially, the condition c is turned into an obligation for all the predecessor program points using weakest preconditions. We attempt to use the abstract state at those points to discharge the condition. This approach is good at handling disjunctive invariants which our abstract domains typically do not represent precisely. E.g., an assert after a join point may not be provable due to loss of precision at the join. However, the abstract states at the program points just prior to the join may be strong enough to discharge the obligation. This backwards analysis discharges an obligation if it can be discharged on *all* the paths leading to the assertion. It thus acts as a form of on-demand trace partitioning [37]. Furthermore, it also provide: (i) another way of modularly combining different analyses, as for instance one branch may be discharged by the non-null analysis and the other by the numerical analysis (the common case for implication-like conditions such as $\mathsf{a} == \texttt{null} \mid\mid \mathsf{a.Length} > 0$); and (ii) to lazily perform loop unrolling.

Example 7. Let us consider the code in Fig.7. Intuitively the assertion holds because the loop is executed at least once. At program point 3, the NonNull analysis infers $\mathsf{sv}(\mathsf{s}) = \texttt{N} \sqcup \texttt{NN} = \top$, and the numerical analysis infers $\mathsf{sv}(\mathsf{i}) = \mathsf{sv}(\mathsf{a}) \wedge \mathsf{sv}(\mathsf{i}) \in [1, +\infty]$. So the direct check cannot prove the assertion. The condition is pushed back to the predecessor program points, 1 corresponding to 0 executions of the loop, and 2 corresponding to > 0 loop iterations. At 2, we know that $\mathsf{sv}(\mathsf{s}) = \texttt{NN}$ from the forward analysis, so this path can be discharged. At 1, we know that $\mathsf{sv}(\mathsf{i}) = 0$, but $\mathsf{sv}(\mathsf{i}) > 0$ at 3 from the forward analysis, hence a contradiction, so the path $1 \rightarrow 3$ is unfeasible, and the condition can be discharged. □

7 Contract Inference

To help the programmer get started with the CodeContracts, Clousot performs some amount of inference, which is either suggested to the user as missing contracts or silently propagated.

Precondition Inference. When a proof obligation cannot be discharged with any of the methods sketched above, Clousot checks if *all* the variables appearing in the condition: (i) existed in the pre-state of the method; and (ii) are unmodified. In this case it suggests a possible precondition. For instance in the example of Fig. 5, Clousot will suggest the two preconditions `input` $! =$ `null` and `output` $! =$ `null`. The precondition is only *suggested* and not inferred as it may be wrong. In the same example, suppose that the code

```
if(input == null) return;
```

was added before the loop, then Clousot would still have suggested `output` $! =$ `null` as precondition, but it would be incorrect, as `output` can perfectly be `null` when `input` is `null`. We have a better and correct solution for the precondition inference problem [11], but have yet to implement it at the time of writing.

Postcondition Inference. Theoretically the postcondition inference problem is simply the projection of the abstract state(s) at the method return point. In practice one must also consider two facts: (i) avoid repeating postconditions already provided by the user; and (ii) produce a minimal set of postconditions. Our postcondition inference algorithm works as follows. First, ask all the analyses to provide known facts at the method return point. Facts should be serialized as Boolean expressions. Second, sort the Boolean expressions according to some heuristic (*e.g.* equalities are more interesting than inequalities). Call the result S. Third, create a product abstract state R abstracting the method postcondition. Fourth, for each fact $s \in S$, check if it is implied by R. If it is not, output s as a postcondition, and assume s in R. The algorithm produces a set of postconditions which fulfills the two requirements above.

Readonly Field Invariant Inference. We have prototyped a static analysis to infer object invariants on readonly fields based on [30].

8 Practical Considerations

To make Clousot practical, we have engineered several solution to improve the user experience.

Adaptive Analysis, Timeouts. We spent a considerable amount of time profiling and optimizing Clousot. However, there are corner cases in which a method analysis can take too long. Single methods can present complex control flow with a lot of join points (several thousands for a single method) or several nested loops

```
var str = ThirdPartyLibrary.GetSomeString();

Contract.Assume(str != null);

/* Without the assumption, Clousot complains str may be null */
if(str.Length > 10) { ... }
```

Fig. 8. Example of using `Assume` to shut off a warning caused by a missing postcondition on third-party code

causing the fixpoint computation to converge too slowly, in particular with relational domains. We have implemented an adaptive analysis, which tries to figure out if the method to analyze is too complex, in which case it analyzes it with cheaper abstract domains. Orthogonally, the fixpoint computation can be aborted when a certain timeout is reached (by default 10 seconds).

Message Prioritization. Clousot has heuristics for sorting the warning messages, trying to report the more relevant ones first. The heuristics assign an initial score I_P to each warning depending on the proof obligation ($P \in \{$`Precondition`, `Postcondition`, `Invariant`, `Assert`, `NonNullobligation`$\ldots$). The initial score is corrected with a reward ρ for the outcome ($\rho(\mathtt{False}) > \rho(\bot) > \rho(\top) \geq 1$, and a penalty δ on the variables in the condition ($\delta(\mathtt{Param}) > \delta(\mathtt{Field}) > \delta(\mathtt{Local}) \geq 1$). Intuitively, a warning on a condition with only locals (where all the information should be known to Clousot) is more likely to be a bug than one on a condition containing only references to parameters (for instance, the code may be missing a precondition). Eventually, a proof obligation of type P, with condition `C` and outcome O is prioritized according the formula $I_P \cdot \rho(O)/(\sum_{\mathtt{v} \in \mathtt{Vars(C)}} \delta(\mathtt{v}))$.

Dealing with False Positives. There are two main reasons for which Clousot reports a false warning: (i) it does not know some external fact (for instance some third-party library methods returns a non-null value); (ii) it is incomplete (as all the static analyses). The user can help Clousot by adding an explicit assumption via `Contract.Assume`.

Clousot will simply believe the condition, and it will not try to check it statically. The condition can be checked at runtime (it behaves as a normal assertion). With the time, assumptions may grow very large in the codebase. Clousot can be instructed to find duplicated assumptions (essentially Clousot tries to prove the assumption, and if it succeeds reports it to the user, otherwise it silently moves on).

Example 8. Let us consider the code snippet if Fig. 8, abstracting the common case of an application using a third-party library without contracts (yet). Without any contract on `GetSomeString`, Clousot will issue a warning for a possible null deference. The programmer, after reading the documentation, convinced himself that the method will never return `null`, and hence decided to add the assumption, hence documenting the fact that the warning has been reviewed, and classified as a false warning. Clousot will then assume it, and it will not issue the warning anymore. When the author of `ThirdPartyLibrary` releases a new version of its library with contracts, then Clousot will inform the user that the assumption is no longer needed. □

If the assumption is not enough to shut off the warning, then the user can mask it via the `SuppressMessage` attribute. This is normally the case when a contract is far beyond what Clousot can understand (for instance it involves several quantifiers). Furthermore, the user can focus the analysis on a particular type or method via the `ContractVerification` attribute.

Visual Studio Integration and Analysis Caching. Clousot is fully integrated into Visual Studio. In a normal run, it runs as a post-build step. Running synchronously the whole analysis at every build may decrease the user experience. As a consequence we have implemented a caching mechanism to re-analyze a small subset of the code that changed between two builds. Orthogonally, the user can make the verification process more interactive by using the "analyze this" feature, which runs the analysis only on the particular method or class under the mouse pointer.

9 Conclusions

We presented an overview of Clousot, a static checked for CodeContracts. Clousot analyzes annotated programs to infer facts (including loop invariants), and it uses this information to discharge proof obligations. Unlike similar tools, it is based on abstract interpretation and focused on specific properties of interest. Advantages include more determinism, (tunable) performance and automation. Clousot is distributed with the CodeContracts tools, available for downloading with academic license at `http://research.microsoft.com/en-us/projects/contracts/`. So far, we have had positive feedback from our users. Still there is much work to do, like increasing the expressivity of the heap analysis, adding abstract domains for strings and bit vectors, improving the inter-method inference, and facilitating the annotation process of legacy codebases.

Acknowledgments. We'd like to thank Mike Barnett and all those who contributed to Clousot: Patrick and Radhia Cousot, Pietro Ferrara, Matthias Perron, Jacques-Henri Jourdan, Vincent Laviron, and Michäel Monerau.

References

1. Bagnara, R., Hill, P.M., Zaffanella, E.: Applications of polyhedral computations to the analysis and verification of hardware and software systems. Theor. Comput. Sci. 410(46) (2009)
2. Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006)

3. Barthe, G., Burdy, L., Charles, J., Grégoire, B., Huisman, M., Lanet, J.-L., Pavlova, M., Requet, A.: JACK — A Tool for Validation of Security and Behaviour of Java Applications. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2006. LNCS, vol. 4709, pp. 152–174. Springer, Heidelberg (2007)
4. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. In: PLDI 2003 (2003)
5. Chen, L., Miné, A., Cousot, P.: A Sound Floating-Point Polyhedra Abstract Domain. In: Ramalingam, G. (ed.) APLAS 2008. LNCS, vol. 5356, pp. 3–18. Springer, Heidelberg (2008)
6. Clarisó, R., Cortadella, J.: The Octahedron Abstract Domain. In: Giacobazzi, R. (ed.) SAS 2004. LNCS, vol. 3148, pp. 312–327. Springer, Heidelberg (2004)
7. Cousot, P., Cousot, R.: Comparing the galois connection and widening/narrowing approaches to abstract interpretation. In: Bruynooghe, M., Wirsing, M. (eds.) PLILP 1992. LNCS, vol. 631, pp. 269–295. Springer, Heidelberg (1992)
8. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: ACM POPL 1979 (1979)
9. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: 4th POPL, pp. 238–252. ACM Press, New York (1977)
10. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: Combination of Abstractions in the ASTRÉE Static Analyzer. In: Okada, M., Satoh, I. (eds.) ASIAN 2006. LNCS, vol. 4435, pp. 272–300. Springer, Heidelberg (2008)
11. Cousot, P., Cousot, R., Logozzo, F.: Contract precondition inference from intermittent assertions on collections. In: VMCAI 2011 (2011)
12. Cousot, P., Cousot, R., Logozzo, F.: A parametric segmentation functor for fully automatic and scalable array content analysis. In: Proceeding of the 38^{th} ACM Symposium on Principles of Programming Languages (POPL 2011). ACM Press, New York (January 2011)
13. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: ACM POPL 1978 (1978)
14. Distefano, D., Matthew, J., Parkinson, J.: jStar: Towards practical verification for Java. In: OOPSLA 2008: Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications, pp. 213–226. ACM, New York (2008)
15. ECMA. Standard ECMA-355, Common Language Infrastructure (June 2006)
16. Fähndrich, M., Barnett, M., Logozzo, F.: Code Contracts (March 2009)
17. Fähndrich, M., Barnett, M., Logozzo, F.: Embedded contract languages. In: ACM SAC 2010 (2010)
18. Ferrara, P., Logozzo, F., Fähndrich, M.: Safer unsafe code in.NET. In: OOPSLA 2008. ACM Press, New York (2008)
19. Filliâtre, J.-C., Marché, C.: The Why/Krakatoa/Caduceus Platform for Deductive Program Verification. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 173–177. Springer, Heidelberg (2007)
20. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for java. In: PLDI 2002 (2002)
21. Gopan, D., Reps, T.W., Sagiv, S.: A framework for numeric analysis of array operations. In: 32nd POPL, pp. 338–350. ACM Press, New York (2005)
22. Gulwani, S., McCloskey, B., Tiwari, A.: Lifting abstract interpreters to quantified logical domains. In: 35th POPL, pp. 235–246. ACM Press, New York (2008)

23. Jacobs, B., Smans, J., Piessens, F.: A quick tour of the veriFast program verifier. In: Ueda, K. (ed.) APLAS 2010. LNCS, vol. 6461, pp. 304–311. Springer, Heidelberg (2010)
24. Jeannet, B., Miné, A.: Apron: A library of numerical abstract domains for static analysis. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 661–667. Springer, Heidelberg (2009)
25. Jhala, R., McMillan, K.L.: Array Abstractions from Proofs. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 193–206. Springer, Heidelberg (2007)
26. Karr, M.: Affine relationships among variables of a program. Acta Inf. 6 (1976)
27. Khachiyan, L., Boros, E., Borys, E., Elbassioni, K.M., Gurvich, V.: Generating all vertices of a polyhedron is hard. Discrete & Computational Geometry 39(1-3), 174–190 (2008)
28. Laviron, V., Logozzo, F.: Refining Abstract Interpretation-Based Static Analyses with Hints. In: Hu, Z. (ed.) APLAS 2009. LNCS, vol. 5904, pp. 343–358. Springer, Heidelberg (2009)
29. Laviron, V., Logozzo, F.: SubPolyhedra: A (More) Scalable Approach to Infer Linear Inequalities. In: Jones, N.D., Müller-Olm, M. (eds.) VMCAI 2009. LNCS, vol. 5403, pp. 229–244. Springer, Heidelberg (2009)
30. Logozzo, F.: Modular static analysis of object-oriented languages. Thèse de doctorat en informatique, École polytechnique (2004)
31. Logozzo, F., Fähndrich, M.: On the Relative Completeness of Bytecode Analysis Versus Source Code Analysis. In: Hendren, L. (ed.) CC 2008. LNCS, vol. 4959, pp. 197–212. Springer, Heidelberg (2008)
32. F. Logozzo and M. Fähndrich. Pentagons: a weakly relational abstract domain for the efficient validation of array accesses. In: ACM SAC 2008 (2008)
33. Miné, A.: A few graph-based relational numerical abstract domains. In: Hermenegildo, M.V., Puebla, G. (eds.) SAS 2002. LNCS, vol. 2477, p. 117. Springer, Heidelberg (2002)
34. Miné, A.: Relational Abstract Domains for the Detection of Floating-Point Run-Time Errors. In: Schmidt, D. (ed.) ESOP 2004. LNCS, vol. 2986, pp. 3–17. Springer, Heidelberg (2004)
35. Miné, A.: The octagon abstract domain. Higher-Order and Symbolic Computation 19, 31–100 (2006)
36. Miné, A.: Symbolic Methods to Enhance the Precision of Numerical Abstract Domains. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 348–363. Springer, Heidelberg (2005)
37. Rival, X., Mauborgne, L.: The trace partitioning abstract domain. ACM Trans. Program. Lang. Syst. 29(5) (2007)
38. Sankaranarayanan, S., Ivančić, F., Gupta, A.: Program Analysis Using Symbolic Ranges. In: Riis Nielson, H., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 366–383. Springer, Heidelberg (2007)
39. Simon, A., King, A., Howe, J.M.: Two variables per linear inequality as an abstract domain. In: Leuschel, M. (ed.) LOPSTR 2002. LNCS, vol. 2664. Springer, Heidelberg (2003)
40. Smans, J., Jacobs, B., Piessens, F.: VeriCool: An Automatic Verifier for a Concurrent Object-Oriented Language. In: Barthe, G., de Boer, F.S. (eds.) FMOODS 2008. LNCS, vol. 5051, pp. 220–239. Springer, Heidelberg (2008)
41. Tillmann, N., de Halleux, J.: Pex–White Box Test Generation for.NET. In: Beckert, B., Hähnle, R. (eds.) TAP 2008. LNCS, vol. 4966, pp. 134–153. Springer, Heidelberg (2008)

Abstract Compilation of Object-Oriented Languages into Coinductive CLP(X): Can Type Inference Meet 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. This paper further investigates the potential and practical applicability of *abstract compilation* in two different directions. First, we formally define an abstract compilation scheme for precise prediction of uncaught exceptions for a simple Java-like language; besides the usual user declared checked exceptions, the analysis covers the runtime `ClassCastException`. Second, we present a general implementation schema for abstract compilation based on coinductive CLP with variance annotation of user-defined predicates, and propose an implementation based on a Prolog prototype meta-interpreter, parametric in the solver for the subtyping constraints.

1 Introduction

Mapping type checking and type inference algorithms to inductive constraint logic programming (CLP) is not a novel idea. Sulzmann and Stuckey [22] have shown that the generalized Hindley/Milner type inference problem HM(X) [19] 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, enhances clarity and modularity of the specification of type inference, 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.

Recent work has shown how coinductive logic programming [21] and coinductive CLP can be fruitfully applied to a handful of applications ranging over type inference of object-oriented languages [6,2,3], verification of real time systems [18], model checking, and SAT solvers [17].

* This work has been partially supported by MIUR DISCO - Distribution, Interaction, Specification, Composition for Object Systems.

Type inference can be defined in terms of *abstract compilation* [6,2,3] into a Horn formula of the program to be analyzed, and of resolution of an appropriate goal in coinductive CLP with subtyping constraints. In contrast to conventional inductive CLP, coinductive CLP allows the specification of much more expressive type systems and, therefore, of more precise forms of type analysis able to better detect the malfunctioning of a program.

We recently discovered that the phrase "abstract compilation" has been introduced to describe a technique for enhancing the performance of abstract interpretation for global flow analysis of logic programs [12], by transforming a program into another program that when executed yields the desired information about the original program. Such a technique is not based on coinductive CLP and was not conceived for type analysis of object-oriented languages.

Abstract compilation is particularly interesting for type inference of object-oriented languages when coinduction, union and object types are combined together. A formal definition of abstract compilation [6,2] has been already given for a purely functional object-oriented language similar to Featherweight Java (FJ) [14] with optional nominal type annotations, generalized explicit constructor declarations and primitive types, but no type casts. The proposed abstract compilation scheme supports precise type inference based on coinductive union and object types, and smoothly integrates it with nominal type annotations, which are managed as additional constraints imposed by the user.

To further investigate the scalability of the approach, we have studied an abstract compilation scheme [5] for a simple Java-like language with imperative features such as variable and field assignment and iterative constructs, by considering as source to abstract compilation an SSA [9] intermediate form. The existence of a natural encoding of φ functions (a notion specific of SSA) with union types is an evidence of how SSA intermediate forms can be fruitfully exploited by abstract compilation.

Though these results show that abstract compilation is attractive and promising, its full potential has not been completely explored yet, and more efforts are required before the approach can be applied to realistic object-oriented languages. In this paper we add a further step towards the long way to real applicability of abstract compilation, in two different directions. First, we consider an important feature in modern mainstream object-oriented language, namely exception handling, and show an abstract compilation scheme allowing precise prediction of uncaught exceptions for a simple Java-like language. Second, we present a general implementation schema for abstract compilation based on coinductive CLP, and propose an implementation based on a Prolog prototype meta-interpreter, parametric in the solver for the subtyping constraints. The implementation exploits variance annotations of user-defined predicates 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 introduces abstract compilation with some examples, whereas Section 4 formally defines abstract compilation for a simple Java-like language with exceptions. Section 5 presents a general implementation schema for abstract compilation and is devoted to the semantics and implementation of coinductive CLP; this section can be read independently from Section 4. Finally, Section 6 draws some conclusions and outlines some directions for further investigation. The formal definition of the abstract compilation and of the main auxiliary predicates can be found in the extended version of this paper [1].

2 Background: Coinductive LP/SLD and CLP(X)

Simon et al [21] 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 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 introduces *constraints* in the body of the clauses of a logic program, specifying conditions under which the clauses hold, and allow 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 Abstract Compilation by Example

This section shows how abstract compilation allows accurate analysis of uncaught exceptions, and informally introduces the main concepts which will be used in the formalization given in Section 4.

The terms of our type domain are class, method and field names (represented by constants), and types coinductively defined over integer, boolean, object, union, and exception types.

$$\begin{array}{lll} bt ::= int \mid bool & & \text{(basic types)} \\ vt ::= bt \mid obj(c, [f_1{:}vt_1, \ldots, f_n{:}vt_n]) \mid vt_1 \vee vt_2 & & \text{(value types)} \\ t\ \ ::= vt \mid t_1 \vee t_2 \mid ex(c) & & \text{(types)} \end{array}$$

An object type $obj(c, [f_1{:}vt_1, \ldots, f_n{:}vt_n])$ specifies the class c to which the object belongs, together with the set of available fields with their corresponding value types. A value type does not contain exception types, and represents a set of values. Exception types are inferred for expressions whose evaluation throws an exception, hence cannot be associated with a field or with the parameter of a method. The class name of the object type 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 [7,13]. Finally, if an expression has type $ex(c)$, then it means that its evaluation throws an exception of class c. In general we expect the type of an expression to be the union of value and exception types; for instance, if an expression has type $ex(c) \vee int$, then it means that its evaluation may either throw an exception of class c, or return an integer value. However, if the type of an expression is the union of sole exception types, then it means that the evaluation of that expression will always throw an exception, thus revealing a problem in the program. This accurate analysis is not possible in the approach of Jo et al. [16].

As pointed out in Section 2, in coinductive logic programming terms and derivations can correspond to arbitrary infinite trees [8], hence not all the terms and derivations can be represented in a finite way, therefore the corresponding type systems are called *idealized*. However, an implementable sound approximation of an idealized type system can be obtained by restricting 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 $X_i = e_i$, where all variables X_i are distinct and expressions e_i may only contain variables X_i [8,21,20].

A *type domain* $\mathcal{D}$ is a constraint domain which defines two predicates: strong equivalence and subtyping. In this example strong equivalence corresponds to syntactic equality (restricted forms of the equivalence induced by the subtyping relation could be considered as well) and is interpreted in the coinductive Herbrand universe, whereas subtyping is interpreted as set inclusion between sets of values: $t_1 \leq t_2$ iff $[\![t_1]\!] \subseteq [\![t_2]\!]$, where $[\![t]\!]$ depends on the considered type language. For space limitation, we have omitted the definition of interpretation for our types; however, the definition given by Ancona and Lagorio [3,4] can be extended in a straightforward way to deal with exception types too.

An Accurate Analysis of Uncaught Exceptions. We show by a simple example how abstract compilation allows accurate uncaught exceptions in Java-like languages. We share the same motivations as in the work by Jo et al. [16]: an analysis of uncaught exceptions independent of declared thrown exceptions is a valuable tool for avoiding unnecessary or too broad declarations, and, hence, unnecessary **try** statements or too general error handling. Last but not least, reporting some kinds of unchecked exceptions, as `ClassCastException`, would allow static detection of typical run-time errors. This last feature is supported in the language defined in Section 4.

Consider the following example of Java code (for clarity we use full Java, even though this example could be easily recast in the language defined in Section 4).

```
class Exc extends Exception {
}
interface Node {Node next() throws Exc; // linked nodes
}
class TNode implements Node { // terminal nodes
    public Node next() throws Exc {throw new Exc();}
}
class NTNode implements Node { // non terminal nodes
    private Node next;
    public NTNode(Node n){this.next=n;}
    public Node next() {return this.next;}
}
class Test {// Exc must be declared but will not been thrown
    void m() throws Exc {
        new NTNode(new NTNode(new TNode())).next().next();}
}
```

In order to be correctly compiled, method `m` must declare `Exc` in its **throws** clause, or its body must be wrapped by a dummy **try** statement, even though such a method will never throw an exception of type `Exc`. The **throws** clause can be safely removed, if a type more precise than `Node` is inferred for the expression **new** `NTNode(`**new** `NTNode(`**new** `TNode())).next()`; indeed, by abstract compilation it is possible to infer the type $obj(ntnode, [next{:}obj(tnode, [\,])])$ and, hence, deduce that the second call to `next()` cannot throw an exception.

To compile the program shown above 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 `next()` of classes `TNode` and `NTNode` generates the following two clauses, respectively.

```
has_meth(tnode,next,[This],ex(exc)).
has_meth(ntnode,next,[This],N) ← field_acc(This,next,N).
```

Predicate *has_meth* has four arguments: the class where the method is declared, the name of the method, the types of the arguments, and the type of the returned value. If a method has n arguments, then its argument type is a list of $n + 1$ types, where the first type always corresponds to the target object **this**. The first clause is a fact specifying that method `next()` declared in class `TNode` always throws an exception. The second clause has a non empty body corresponding to the abstract compilation of the body of the method: `field_acc(This,next,N)` means that accessing field `next` of the object `This` returns a value of type `N`.

Each method declaration is compiled into a clause defining predicate *has_meth*, and, analogously, each constructor declaration is compiled into a clause defining predicate *has_constr*. Furthermore, other program independent clauses are generated to specify the behavior of the various constructs w.r.t. the available types (see Section 4).

Coinductive Derivations and Subtyping. To see an example of coinductive derivation and to explain the importance of subtyping constraints, let us add the following factory method to class `Test` (this is just a simple example in our functional Java-like language; in Java the method would be static and tail recursion would be replaced with a loop).

```
Node addNodes(int i, Node n) { // adds i nodes before n
    if(i<=0) return n;
    else return addNodes(i-1,new NTNode(n));}
```

Let us assume now that we would like to infer the type of the expression `new Test().addNodes(5,new TNode())`; the inferred type can be obtained by resolving the goal $invoke(obj(test, [\,]), addNodes, [int, obj(tnode, [\,])], R_0)$ w.r.t. the Horn formula obtained from the abstract compilation of our example classes.

If we consider unification with no subtyping constraints, then we can only get an infinite derivation containing the following sequence of atoms:

$$\begin{array}{c} at_0 = invoke(obj(test,[\,]), addNodes, [int, t_0], R_0) \\ at_1 = invoke(obj(test,[\,]), addNodes, [int, t_1], R_1) \\ \vdots \\ at_k = invoke(obj(test,[\,]), addNodes, [int, t_k], R_k) \\ \vdots \end{array}$$

with answer $R_0 = t_0 \vee R_1, \ldots, R_k = t_k \vee R_{k+1}, \ldots$, where $t_0 = obj(tnode, [\,])$, and $t_{k+1} = obj(ntnode, [n{:}t_k])$ for all $k \geq 0$; hence, the solution is a non regular term obtained from a non regular derivation. The main problem is that for all k, atom at_k does not unify with atoms $at_0, \ldots, at_{k-1}$, hence no coinductive hypothesis can be used to build a regular proof.

If we consider subtyping and observe that method invocation is contravariant in the argument type and covariant in the returned type, then we have that our initial goal succeeds if the atom $at = invoke(obj(test, [\,]), addNodes, [int, T], R_0)$ succeeds, and $t_0 \leq T$ holds (the resolution steps have been slightly simplified for the sake of clarity). To resolve at, the following atom at' needs to be resolved, under the constraint $R_0 \geq T \vee R_1$:

$$invoke(obj(test, [\,]), addNodes, [int, obj(ntnode, [n{:}T])], R_1)$$

To derive at' we can use the coinductive hypothesis at if the additional constraints $R_1 \geq R_0$ and $obj(ntnode, [n{:}T]) \leq T$ hold. Hence, the initial goal can be resolved if the following set of constraints is satisfiable:

$$t_0 \leq T, R_0 \geq T \vee R_1, R_1 \geq R_0, obj(ntnode, [n{:}T]) \leq T.$$

A possible solution is given by $R_0 = R_1 = T, T = t$, where t is the regular term t s.t. $t = t_0 \vee obj(ntnode, [n{:}t])$. Therefore, by exploiting the subtyping constraint

$\leq$, we can resolve our goal with a regular derivation and a regular solution (other interesting examples of regular derivations, which can be computed by considering the subtyping constraint, can be found in related papers [2,3,4]).

The derivation sketched above follows the rules of coinductive CLP to be defined in Section 5, where user-defined predicates are associated with *variance annotations*.

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 argument (the definition can be straightforwardly generalized for an arbitrary number of arguments), 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 as well (we say that $p(t_1)$ subsumes $p(t_2)$).
– p is *contravariant* in its argument: if $t_2 \leq t_1$ 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 and second arguments, contravariant in its third argument, and covariant in its fourth argument. Note that, in contrast with what intuition may suggest, weak invariance in the first argument of *invoke* is unsound.

4 Formalization

In this section we formally define abstract compilation for a simple functional Java-like language supporting exceptions (Fig. 1). Syntactic assumptions listed in the figure have to be verified before abstract compilation is performed. Notation as $\overline{cd}^n$ denote sequences of n items.

$$
\begin{array}{rcl}
\mathit{prog} & ::= & \overline{cd}^n\ e \\
\mathit{cd} & ::= & \texttt{class}\ c_1\ \texttt{extends}\ c_2\ \{\ \overline{fd}^n\ \mathit{cn}\ \overline{md}^k\ \}\quad (c_1 \neq \mathit{Object}, \mathit{Throwable}, \mathit{ClassCastExc}) \\
\mathit{fd} & ::= & \tau\ f; \\
\mathit{cn} & ::= & c(\overline{\tau\ x}^n)\ \{\texttt{super}(\overline{e}^k); \overline{f = e';}^h\} \\
\mathit{md} & ::= & \tau_0\ m(\overline{\tau\ x}^n)\ \{e\} \\
e & ::= & \texttt{new}\ c(\overline{e}^n) \mid x \mid e.f \mid e_0.m(\overline{e}^n) \mid \texttt{if}\ (e)\ e_1\ \texttt{else}\ e_2 \mid \texttt{false} \mid \texttt{true} \mid i \mid e_1\ \mathit{op}\ e_2 \\
 & & \texttt{throw}\ c \mid \texttt{try}\ e_1\ \texttt{catch}(c)\ e_2 \mid (c)\ e \\
\mathit{op} & ::= & \mathit{relOp} \mid \mathit{boolOp} \mid \mathit{intOp} \\
\tau & ::= & c \mid \texttt{bool} \mid \texttt{int}
\end{array}
$$

Assumptions: $n, k, h \geq 0$, inheritance is acyclic, names of declared classes in a program, methods and fields in a class, and parameters in a method are distinct.

Fig. 1. Syntax of the language

A program consists of a sequence of class declarations and a main expression. Type annotations in all declarations can be either primitive types *bool* or *int*, or class names. We assume that the language supports boxing conversions, hence *bool* and *int* are both subtypes of *Object*. Hence *Object* is the top type annotation which, in fact, does not impose any restriction on the type of fields, parameters and returned values.

A class declaration contains field and method declarations, and a single constructor declaration. We assume predefined classes *Object*, *Throwable* and *ClassCastExc*: the first is the root of the inheritance tree, the second extends *Object* and is the most general type for exceptions, the third extends *Throwable* and is the class of the unchecked exceptions thrown when a runtime type check fails; for simplicity, we assume that all three classes contain no fields and methods and have a constructor without parameters. The body of a constructor consists of an invocation of the superclass constructor and a sequence of field initializations, one for each field declared in the class. Method declarations are standard; note however that they do not include **throws** clause, since our analysis is independent of the declared thrown exceptions.

Expressions deserve few comments: i denotes integer literals, *relOp*, *boolOp* and *intOp* denotes the usual relational, boolean and integer binary operators; for simplicity, we consider `==` and `!=` monomorphic operators over integers; an extension allowing `==` and `!=` to be polymorphic is straightforward. Expressions for exception handling have been deliberately simplified to make the presentation lighter: when an exception is thrown, no instance is created, but only the type of the exception (which is required to be a subtypes of *Throwable*) is specified; consequently, catch clauses do not have any formal parameter. Furthermore **try** expressions can have only one catch clause.

For space reasons we have omitted the quite standard operational semantics of the language. The abstract compilation for programs, declarations, and expressions can be found in the extended version of this paper [1]. Abstract compilation of a program generates a pair $(Hf|B)$, where Hf is a Horn formula and B is a goal (a sequence of atoms). Abstract compilation of a class, field, constructor, and method declaration yields two clauses (for classes and methods) or one (for fields and constructors).

For simplicity class, field, method and variable names are not affected by abstract compilation, even though in practice appropriate bijections (different from the identity) have to be considered. This is due to the fact that in logic programming names beginning with an upper case letter denote logical variables, while those beginning with a lower case letter denote constant, function and predicate symbols.

For any expression e, the abstract compilation of e generates a pair $(t|B)$, where t is the term corresponding to the type of e, and B is the sequence of atoms whose satisfaction ensures that e is well-typed. The compilation is straightforward and is based on a set of predicates which specify the behavior of each construct. For instance, predicate *invoke* is defined as follows:

```
invoke(obj(C,R),M,A1,RT∨ET) ← val_types(A1,A2),
           exc_types(A1,ET),has_meth(C,M,[obj(C,R)|A2],RT).
invoke(obj(C,R),M,A,ET) ← no_val_types(A), exc_types(A,ET).
invoke(T1∨T2,M,A,RT1∨RT2) ← invoke(T1,M,A,RT1),
                             invoke(T2,M,A,RT2).
invoke(ex(C),M,A,ex(C)).
```

The first two clauses specify the behavior of method calls when the target is an object type. The predicates `val_types`, `exc_types`, and `no_val_types` (whose definition can be found in the extended version of this paper [1]) control exception propagation during argument evaluation. The atom `val_types(`l`,`l'`)` succeeds only if type list l corresponds to an expression sequence whose evaluation may be completed normally (see Section 14.1, [11]) with type list l' (which necessarily contains no exception types). For instance, `val_types([`$ex(c_1) \vee vt_1$`,`$ex(c_2) \vee vt_2$`],[`vt_1`,`vt_2`])` succeeds, whereas `val_types([`$ex(c_1)$`,`$ex(c_2) \vee vt_2$`],X)` fails. The atom `exc_types(`l`,`t`)` succeeds if l corresponds to an expression sequence whose evaluation may be completed abruptly with type t (which necessarily does not contain value types). For instance, `exc_types([`$ex(c_1) \vee vt_1$`,`$ex(c_2) \vee vt_2$`],`$ex(c_1) \vee ex(c_2)$`)` succeeds; note that `exc_types` succeeds also when no exceptions are thrown: `exc_types([`vt_1`,`vt_2`],X)` succeeds with `X=`$\perp$ (that is, the empty type, which can be simply represented by the type t s.t. $t = t \vee t$ [3,4]). Finally, `no_val_types(`l`)` succeeds iff `val_types(`l`,X)` fails.

The first clause of *invoke* deals with cases where argument expressions may evaluate normally. Method look-up is started (predicate `has_meth`) from the class of the target object, and its type is added as first argument to correctly deal with **this**. Note that this case does not prevent argument expressions to evaluate abruptly: `ET` represents all thrown exceptions. The second clause is used when argument expressions never evaluate normally: in this case no method look-up is performed[1]; this clause allows exact propagation of union types containing sole exception types, thus inferring that the evaluation of the method invocation will always throw an exception, and, hence, that something is wrong in the source code.

The third clause of *invoke* deals with union types: if invoking method `M` on a target of type `T1` (resp. `T2`) yields a result of type `RT1` (resp. `RT2`), then invoking `M` on a target of type `T1` ∨ `T2` yields a result of type `RT1` ∨ `RT2`.

Finally, the last clause deals with the case when the expression corresponding to the target evaluates abruptly by throwing an exception `exc(C)` which, therefore, is propagated.

Let us focus now on the predicates corresponding to the **throw** and **try** constructs. The clause for *throw* is pretty straightforward:

```
throw(C,ex(C)) ← subtype(C,throwable).
```

The type of the expression **throw**`(`c`)` is $ex(c)$, providing that c is a subtype of *Throwable*, otherwise the expression is not well-typed.

[1] This allows typechecking of an invocation of *any* method `M` with *any* arguments `A`, though the method will never be actually invoked, since the evaluation of its arguments will *always* throw an exception.

Since the behavior of the **try** expression is more involved, let us consider first some examples. If e_1 and e_2 have type $t_1 = ex(c_1) \vee ex(c_2) \vee vt$ and t_2, respectively, and if c_1 is a subclass of c, while c_2 is not, then the type inferred for `try` e_1 `catch`(c) e_2 is $ex(c_2) \vee vt \vee t_2$. On the other hand, if both c_1 and c_2 are not subclasses of c, then the inferred type is just t_1. Indeed, expression e_2 is evaluated only if e_1 throws an exception which is a subclass of c, hence the type of the **try** expression includes t_2 only when t_1 contains an exception type handled by the **catch** clause. Furthermore, all exception types in t_1 handled by the **catch** clause have to be removed from t_1 to infer the most precise type.

```
try(T1,C,T2,T3∨T2) ← remove_handled(T1,C,T3).
try(T1,C,T2,T1) ← unhandled(T1,C).
```

The auxiliary predicate `remove_handled(T1,C,T3)` succeeds if `T1` contains at least an exception type covered by `C`, and the type `T3` is obtained from `T1` by removing all exception types covered by `C`; the auxiliary predicate `unhandled(T1,C)` succeeds if `T1` does not contain an exception type covered by `C`. The complete definition of all main and auxiliary predicates can be found in the extended version of this paper [1].

5 A Prototype Implementation of Coinductive CLP(X)

In this section we show a prototype implementation of the inference engine for coinductive CLP, which is an essential component for supporting abstract compilation, as depicted in Fig. 2. The input is represented by the source program to be analyzed and by a query defined by the user in a high level language. Then the abstract compiler and the goal generator, which is a subcomponent of the abstract compiler, generate a Horn formula 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 (if the source language is unchanged only the back-end will be modified) and the solver for the specific type domain.

The engine supports variance annotations, which are more than a convenient syntactic notation for avoiding explicit insertion of constraints in the body of

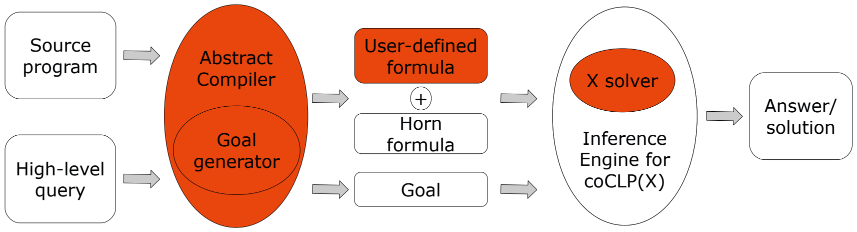


Fig. 2. General schema for abstract compilation based on coinductive CLP(X)

clauses; indeed, by associating constraints with predicates, expressive power is enhanced since it is possible to exploit subsumption, instead of plain unification, when applying the coinductive hypothesis rule. To our knowledge, this is a novel feature not previously considered in CLP.

We first provide the fixed-point and operational semantics of coinductive CLP.

Fixed-Point Semantics. For simplicity, all following definitions use a fixed coinductive Herbrand universe and base and type domain $\mathcal{D}$.

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 constraint predicates $\{\leq, \geq, \cong, \equiv\}$, as defined in Sect. 3.

Definition 1. *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, $\mathcal{D} \models \{t_1 \alpha_1 t'_1, \ldots, t_n \alpha_n t'_n\}$.*

The one-step consequence function $T_{\mathit{Hf},\mathcal{D}}$, 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:

$$\begin{aligned} T_{\mathit{Hf},\mathcal{D}}(S) = \{A' \mid\ & A \leftarrow A_1, \ldots, A_n \text{ ground instance of a clause in } \mathit{Hf}, \\ & A_i \in S \text{ for all } i = 1, \ldots, n,\ A \text{ subsumes } A'\} \end{aligned}$$

The coinductive Herbrand model of Hf w.r.t. the type domain $\mathcal{D}$ is the greatest fixed-point of $T_{\mathit{Hf},\mathcal{D}}$. Equivalently, the fixed-point 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 $T^{CLP}_{\mathit{Hf}',\mathcal{D}}$, where $T^{CLP}_{\mathit{Hf}',\mathcal{D}}$ is the standard one-step consequence function defined for CLP [15]:

$$\begin{aligned} T^{CLP}_{\mathit{Hf},\mathcal{D}}(S) = \{A \mid\ & A \leftarrow C, A_1, \ldots, A_n \text{ ground instance of a clause in } \mathit{Hf}, \\ & A_i \in S \text{ for all } i = 1, \ldots, n,\ \mathcal{D} \models C\} \end{aligned}$$

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:

$$\begin{aligned} & \mathit{gen}(\epsilon, \epsilon, \epsilon) = \emptyset \\ & \mathit{gen}((t, \overline{t}^{n-1}), (\alpha, \overline{\alpha}^{n-1}), (u, \overline{u}^{n-1})) = \{t\, \alpha\, u\} \cup \mathit{gen}(\overline{t}^{n-1}, \overline{\alpha}^{n-1}, \overline{u}^{n-1}) \end{aligned}$$

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.

$$\text{(empty)}\quad \mathit{Hf} \mid H \vdash \mathit{true} \rightsquigarrow \emptyset$$

$$\text{(co-hyp)}\frac{\mathit{Hf} \mid H \vdash G_1, G_2 \rightsquigarrow C_1 \qquad \vdash C_1 \cup C_2 \rightarrow C'}{\mathit{Hf} \mid H \vdash G_1, p_{\overline{\alpha}^n}(\overline{u}^n), G_2 \rightsquigarrow C'} \quad \begin{array}{l} C_2 = \mathit{gen}(\overline{t}^n, \overline{\alpha}^n, \overline{u}^n) \\ H = H_1, p_{\overline{\alpha}^n}(\overline{t}^n), H_2 \end{array}$$

$$\text{(cls)}\frac{\mathit{Hf} \mid H, p_{\overline{\alpha}^n}(\overline{t}^n) \vdash G_1, G, G_2 \rightsquigarrow C_1 \qquad \vdash C_1 \cup C_2 \rightarrow C'}{\mathit{Hf} \mid H \vdash G_1, p_{\overline{\alpha}^n}(\overline{u}^n), G_2 \rightsquigarrow C'} \quad \begin{array}{l} p_{\overline{\alpha}^n}(\overline{t}^n) \leftarrow G \text{ fresh} \\ \text{instance of a} \\ \text{clause of } \mathit{Hf} \\ C_2 = \mathit{gen}(\overline{t}^n, \overline{\alpha}^n, \overline{u}^n) \end{array}$$

Fig. 3. Operational semantics

Operational Semantics. The operational semantics of a Horn formula Hf is inductively defined in Fig. 3.

In the judgment $\mathit{Hf} \mid H \vdash G \rightsquigarrow C$, meta-variables Hf, H, and G represent the input of the judgment, whereas C is the only output; if the judgment is derivable, then the goal G succeeds w.r.t. the Horn formula Hf and the coinductive hypotheses H, with the satisfiable set of constraints C as solution.

The coinductive hypotheses H (a stack of atoms) are needed 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).

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 from 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 $\mathit{gen}(\overline{t}^n, \overline{\alpha}^n, \overline{u}^n)$ 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, then 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 Fig. 3 with a meta-interpreter[2] written in SWI Prolog. The implementation performs a depth first search of the tree of 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, rule (co-hyp) takes the precedence over (cls), and 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(($p_{\overline{\alpha}^n}(\overline{u}^n)$), Goal), Solver, CoHyp, C1, Solution) ←
 fresh_atom($p$, $n$, $p_{\overline{\alpha}^n}(\overline{X}^n)$)),member($p_{\overline{\alpha}^n}(\overline{X}^n)$, CoHyp),
 gen($\overline{X}^n$, $\overline{\alpha}^n$, $\overline{u}^n$, C2), union(C1, C2, C3), call(Solver, C3, C4),
 coCLP(Goal, Solver, CoHyp, C4, Solution).
% (cls)
coCLP(($p_{\overline{\alpha}^n}(\overline{u}^n)$), Goal), Solver, CoHyp, C1, Solution) ←
 fresh_atom($p$, $n$, $p_{\overline{\alpha}^n}(\overline{X}^n)$)), clause($p_{\overline{\alpha}^n}(\overline{X}^n)$, Body),
 gen($\overline{X}^n$, $\overline{\alpha}^n$, $\overline{u}^n$, C2), union(C1, C2, C3), call(Solver, C3, C4),
 append_goal(Body, Goal, NewGoal),
 coCLP(NewGoal, Solver, [$p_{\overline{\alpha}^n}(\overline{X}^n)$|CoHyp], C4, Solution).
```

The main predicate (we assume that the goal is always terminated by *true*) `coCLP/3` (not specified in Fig. 3) is defined in terms of the auxiliary predicate `coCLP/5` which implements the judgment $Hf \mid H \vdash G \leadsto C$. 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 tail-recursive, 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

[2] Available at `ftp://ftp.disi.unige.it/person/AnconaD/coCLP.zip`

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.

Further details of the implementation are available in an technical report [1].

6 Conclusion

This paper provides a further step towards applicability of abstract compilation to realistic object-oriented languages in two directions.

We have defined a formal abstract compilation scheme allowing precise prediction of uncaught exceptions for a simple Java-like language. The analysis covers both user declared checked exceptions, and the unchecked predefined runtime exception `ClassCastException`. Furthermore, we have presented a general implementation schema for abstract compilation based on coinductive CLP with variance annotation of user-defined predicates, and proposed an implementation based on a Prolog prototype meta-interpreter, parametric in the solver for the subtyping constraints.

Our approach seems particularly promising in the context of object-oriented programming, when the type domain contains union and object types. More efforts are required to obtain results for realistic object-oriented languages. Devising a constraint solver for subtyping on regular union and object types is of paramount importance. We have already investigated several sound but not complete axiomatizations of subtyping [3,4], but we still do not know whether subtyping on regular union and object types is decidable; currently, we are developing a CHR [10] based implementation of a sound but not complete constraint solver for the abstract compilation scheme presented in this paper. Although scalability of the approach in the presence of imperative features has been already investigated [5], much work should be accomplished in this direction; for instance, it would be interesting to investigate whether abstract compilation could be integrated with other kinds of analysis to detect reference aliasing, or other runtime exceptions as `NullPointerException` or `IndexOutOfBoundsException`.

Acknowledgments. We warmly thank the anonymous FoVeOOS 2010 referees for their useful comments and suggestions.

References

1. Ancona, D., Corradi, A., Lagorio, G., Damiani, F.: Abstract compilation of object-oriented languages into coinductive CLP(X): can type inference meet verification (extended version). Technical report, DISI (August 2010), `ftp://ftp.disi.unige.it/person/AnconaD/ACLD10ext.pdf`
2. Ancona, D., Lagorio, G.: Coinductive type systems for object-oriented languages. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 2–26. Springer, Heidelberg (2009); Best paper prize
3. Ancona, D., Lagorio, G.: Coinductive subtyping for abstract compilation of object-oriented languages into Horn formulas. In: GandALF 2010. Electronic Proceedings in Theoretical Computer Science (2010)

4. Ancona, D., Lagorio, G.: Complete coinductive subtyping for abstract compilation of object-oriented languages. In: 12th Intl. Workshop on Formal Techniques for Java-like Programs, ACM Digital Library (2010)
5. Ancona, D., Lagorio, G.: Idealized coinductive type systems for imperative object-oriented programs. Technical report, DISI, Submitted for journal publication (January 2010)
6. Ancona, D., Lagorio, G., Zucca, E.: Type inference by coinductive logic programming. In: Berardi, S., Damiani, F., de'Liguoro, U. (eds.) TYPES 2008. LNCS, vol. 5497, pp. 1–18. Springer, Heidelberg (2009)
7. Barbanera, F., Dezani-Cincaglini, M., de Liguoro, U.: Intersection and union types: Syntax and semantics. Information and Computation 119(2), 202–230 (1995)
8. Courcelle, B.: Fundamental properties of infinite trees. TCS 25, 95–169 (1983)
9. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. ACM TOPLAS 13, 451–490 (1991)
10. Frühwirth, T.: Constraint Handling Rules. Cambridge University Press, Cambridge (2009)
11. Gosling, J., Joy, B., Steele, G.L., Bracha, G.: The Java language specification, 3rd edn. The Java series. Addison-Wesley, Reading (2005)
12. Hermenegildo, M., Warren, R., Debray, K.: Global flow analysis as a practical compilation tool. J. Log. Program. 13(4), 349–366 (1992)
13. Igarashi, A., Nagira, H.: Union types for object-oriented programming. Journ. of Object Technology 6(2), 47–68 (2007)
14. Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight Java: a minimal core calculus for Java and GJ. ACM TOPLAS 23(3), 396–450 (2001)
15. Jaffar, J., Maher, M.J.: Constraint logic programming: A survey. J. Log. Program. 19(20), 503–581 (1994)
16. Jo, J., Chang, B., Yi, K., Choe, K.: An uncaught exception analysis for Java. Journal of Systems and Software 72(1), 59–69 (2004)
17. Min, R., Gupta, G.: Coinductive logic programming and its application to boolean sat. In: FLAIRS Conference (2009)
18. Saeedloei, N., Gupta, G.: Verifying complex continuous real-time systems with coinductive CLP(R). In: Dediu, A.-H., Fernau, H., Martín-Vide, C. (eds.) LATA 2010. LNCS, vol. 6031, pp. 536–548. Springer, Heidelberg (2010)
19. Odersky, M., Sulzmann, M., Wehr, M.: Type inference with constrained types. Theory and Practice of Object Systems 5(1), 35–55 (1999)
20. Simon, L., Bansal, A., Mallya, A., Gupta, G.: Co-logic programming: Extending logic programming with coinduction. In: Arge, L., Cachin, C., Jurdziński, T., Tarlecki, A. (eds.) ICALP 2007. LNCS, vol. 4596, pp. 472–483. Springer, Heidelberg (2007)
21. Simon, L., Mallya, A., Bansal, A., Gupta, G.: Coinductive logic programming. In: Etalle, S., Truszczyński, M. (eds.) ICLP 2006. LNCS, vol. 4079, pp. 330–345. Springer, Heidelberg (2006)
22. Sulzmann, M., Stuckey, P.J.: HM(X) type inference is CLP(X) solving. Journ. of Functional Programming 18(2), 251–283 (2008)

Validating Timed Models of Deployment Components with Parametric Concurrency*

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

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

Abstract. Many software systems today are designed without assuming a fixed underlying architecture, and may be adapted for sequential, multicore, or distributed deployment. Examples of such systems are found in, e.g., software product lines, service-oriented computing, information systems, embedded systems, operating systems, and telephony. Models of such systems need to capture and range over relevant deployment scenarios, so it is interesting to lift aspects of low-level deployment concerns to the abstraction level of the modeling language. This paper proposes an abstract model of deployment components for concurrent objects, extending the Creol modeling language. The deployment components are parametric in the amount of concurrency they provide; i.e., they vary in processing resources. We give a formal semantics of deployment components 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 concurrent 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 [25]; 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 cores; 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

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

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 [29]. In this paper, we apply performance analysis to formal models 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 [11,19], a modeling language for distributed concurrent objects which communicate by asynchronous method calls and futures. Creol's operational semantics is given in rewriting logic [21] and is executable on Maude [10]. Concurrent objects are reminiscent of Actors [1] 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., [7]), and been integrated with, e.g., Java [28] and Scala [14]. Concurrent objects support compositional verification of concurrent software [2,11], in contrast to multi-threading. 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 [26].

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 intervals 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 Creol extended with time and deployment components. Sect. 6 presents testing and simulation results in the context of the example, Sect. 7 discusses related work, and Sect. 8 concludes the paper.

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

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

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

details see, e.g., [11,19]). Objects are dynamically created instances of classes, their 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, 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 declared fields.) 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 $x := rhs$, sequential composition $s_1; s_2$, **skip**, **if**, **while**, and **return** e constructs are standard. The statement **release** unconditionally releases the processor by suspending the active process. In contrast, the guard g controls processor release in the statement **await** g, and consists of Boolean expressions b over attributes and return tests $x?$ (see below). If g evaluates to false, the current process is *suspended* and the active process 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 rhs include declared variables x, Boolean expressions b, and object creation **new** $C(\overline{e})$. The reserved read-only variable **this** refers to the object identifier and **now** to the current clock value (explained below). Note that pure expressions are denoted by e and 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 here, 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. A future variable refers to a return value which may still need 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 suspends execution of the process, but allows other processes to run. 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*.

Time. In this paper we work with an extended version of the Creol language which includes an implicit time model [6], comparable to a system clock which updates every n milliseconds (representing a time interval). In this extension 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. From an object's local perspective, the passage of time is indirectly observable via **await** statements in this model of timed behavior, and time is advanced when no other activity may occur. In this paper this model of time is used to handle the amount of concurrent activity allowed within a time interval in order to model resource constrains for different deployment scenarios.

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 modeling 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. Resource consumption is captured by subtraction, where $\omega - n = \omega$. Within a time interval, 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 interval by repeating this procedure.

In the modeling 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 the 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, they must 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 the C object will reside in 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.

```
 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

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 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 will show how to run this example on a deployment component.

5 Operational Semantics

The operational semantics of timed Creol with deployment components is presented as a transition system in an SOS style [24]. Transition rules apply to subsets of configurations (the standard context rules are not listed). For simplicity we assume that configurations can be reordered to match the left hand side of the rules (i.e., matching is modulo associativity and commutativity as in rewriting logic [21]). A run of the system is a possibly nonterminating sequence of rule applications. When auxiliary functions are used in the semantics, these are evaluated in between the application of transition rules in a run.

$$
\begin{array}{rcl}
cn & ::= & \epsilon \mid comp \mid object \mid msg \\
 & & \mid future \mid cn\ cn \\
object & ::= & ob(o, C, a, p, q) \\
q & ::= & \emptyset \mid p \mid q \backslash p \mid enqueue(p, q) \\
p & ::= & \{l|s\} \mid select(q, a, cn, t) \mid \text{idle} \\
v & ::= & o \mid f \mid \text{null} \mid b \mid t
\end{array}
\qquad
\begin{array}{rcl}
tcn & ::= & clock\ cn \\
comp & ::= & dc(n, r, u) \\
msg & ::= & invoc(o, f, m, \overline{v}) \\
future & ::= & fut(f, v) \\
clock & ::= & cl(t)
\end{array}
$$

Fig. 3. The syntax for timed runtime configurations

Timed runtime timed configurations tcn, given in Fig. 3, include one global clock and an untimed configuration cn; i.e., a set which consists of deployment components, objects, invocation messages, and futures. The associative and commutative union operator on configurations is denoted by whitespace and the empty configuration by ε. These configurations live inside curly brackets; in the term $\{tcn\}$, tcn captures the entire configuration. The global *clock* is a term $cl(t)$ where t is the current time. A *deployment component* is a term $dc(n, r, u)$ where n is the identifier of the component, r the (non-negative) number of available computing resources, and u the maximum number of resources which can be consumed before the clock advances. An *object* is a term $ob(o, C, a, p, q)$ where o is the object's identifier and C its class, a is an attribute mapping representing the object's fields, p is an *active process*, and q is a *queue of suspended processes*. In the fields a of an object o, the reserved field 'mycomp' is bound to the deployment component associated with o. A process p consists of a mapping l of local variable bindings and a list s of statements, and will be written as $\{l|s\}$ when convenient. An *invocation message* is a term $invoc(o, f, m, \overline{v})$ where o is the callee, f the future to which the call's result shall be returned, m the method name, and $\overline{v}$ lists the call's actual parameter values. A *future* is a term $fut(f, v)$ with identifier f and reply value v (which is $\perp$ when the future's reply value has not been received). Values are object and future identifiers, Boolean expressions, clock values, and null (as well as expressions in the functional language).

Evaluating Expressions. Given a substitution σ, a time t, and a configuration cn, denote by $[\![e]\!]^{cn}_{\sigma,t}$ a confluent and terminating reduction system which reduces expressions e to data values. Let $[\![\textbf{now}]\!]^{cn}_{\sigma,t} = t$. Let $[\![x?]\!]^{cn}_{\sigma,t} = \text{true}$ if $[\![x]\!]^{cn}_{\sigma,t} = f$ and $fut(f, v) \in cn$ for some value $v \neq \perp$, otherwise $[\![x?]\!]^{cn}_{\sigma,t} = \text{false}$. The remaining cases are fairly straightforward, looking up values for declared variables in σ. The reduction of an expression always happens in the context of a given process, object state, and configuration. Thus, $\sigma = a \circ l$ (the composition of the fields a and the local variable bindings l), t is the current global time, and cn the current untimed 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. 4. If a and l are mappings, denote by $\text{dom}(a)$ the domain of a; by $a(x)$ the value bound to x in a (assuming that $x \in \text{dom}(a)$); by $a[x \mapsto v]$ the extension of a in which x is bound to v (and $a[x \mapsto v](x') = a(x')$ if $x \neq x'$); and by $a \circ l$ the composed mapping in which $a \circ l(x) = l(x)$ if $x \in \text{dom}(l)$ (and $a \circ l(x) = a(x)$ if $x \notin \text{dom}(l)$). For simplicity, classes are not represented explicitly in the semantics, but may be seen as static

(skip)

$$\frac{a(\text{mycomp}) = n \quad r > 0}{\begin{array}{c} ob(o, C, a, \{l|\text{skip}; s\}, q)\ dc(n, r, u) \\ \rightarrow ob(o, C, a, \{l|s\}, q)\ dc(n, r-1, u) \end{array}}$$

(Bind-Mtd)

$$\frac{p' = \text{bind}(o, f, m, \overline{v}, C)}{\begin{array}{c} ob(o, C, a, p, q)\ invoc(o, f, m, \overline{v}) \\ \rightarrow ob(o, C, a, p, \text{enqueue}(p', q)) \end{array}}$$

(assign1)

$$\frac{\begin{array}{c} x \in \text{dom}(l) \quad v = [\![e]\!]^{\varepsilon}_{(a \circ l), t} \\ a(\text{mycomp}) = n \quad r > 0 \end{array}}{\begin{array}{c} ob(o, C, a, \{l|x := e; s\}, q) \\ dc(n, r, u)\ cl(t) \\ \rightarrow ob(o, C, a, \{l[x \mapsto v]|s\}, q) \\ dc(n, r-1, u)\ cl(t) \end{array}}$$

(Async-Call)

$$\frac{\begin{array}{c} o' = [\![e]\!]^{\varepsilon}_{(a \circ l), t} \quad \overline{v} = [\![\overline{e}]\!]^{\varepsilon}_{(a \circ l), t} \\ \text{fresh}(f) \quad a(\text{mycomp}) = n \quad r > 0 \end{array}}{\begin{array}{c} ob(o, C, a, \{l|x := e!m(\overline{e}); s\}, q) \\ dc(n, r, u)\ cl(t) \\ \rightarrow ob(o, C, a, \{l|x := f; s\}, q)\ dc(n, r-1, u) \\ cl(t)\ invoc(o', f, m, \overline{v})\ fut(f, \bot) \end{array}}$$

(assign2)

$$\frac{\begin{array}{c} x \in \text{dom}(a) \quad v = [\![e]\!]^{\varepsilon}_{(a \circ l), t} \\ a(\text{mycomp}) = n \quad r > 0 \end{array}}{\begin{array}{c} ob(o, C, a, \{l|x := e; s\}, q) \\ dc(n, r, u)\ cl(t) \\ \rightarrow ob(o, C, a[x \mapsto v], \{l|s\}, q) \\ dc(n, r-1, u)\ cl(t) \end{array}}$$

(return)

$$\frac{\begin{array}{c} v = [\![e]\!]^{\varepsilon}_{(a \circ l), t} \quad r > 0 \\ a(\text{mycomp}) = n \quad l(\text{destiny}) = f \end{array}}{\begin{array}{c} ob(o, C, a, \{l|\text{return } e; s\}, q)\ cl(t) \\ fut(f, \bot)\ dc(n, r, u) \\ \rightarrow ob(o, C, a, \{l|s\}, q)\ cl(t) \\ fut(f, v)\ dc(n, r-1, u) \end{array}}$$

(cond1)

$$\frac{[\![e]\!]^{\varepsilon}_{(a \circ l), t}}{\begin{array}{c} ob(o, C, a, \{l|\text{if } e \text{ then } s_1 \\ \text{else } s_2 \text{ fi}; s\}, q)\ cl(t) \\ \rightarrow ob(o, C, a, \{l|s_1; s\}, q)\ cl(t) \end{array}}$$

(await1)

$$\frac{[\![g]\!]^{cn}_{(a \circ l), t}}{\begin{array}{c} \{ob(o, C, a, \{l|\text{await } g; s\}, q)\ cl(t)\ cn\} \\ \rightarrow \{ob(o, C, a, \{l|s\}, q)\ cl(t)\ cn\} \end{array}}$$

(cond2)

$$\frac{\neg[\![e]\!]^{\varepsilon}_{(a \circ l), t}}{\begin{array}{c} ob(o, C, a, \{l|\text{if } e \text{ then } s_1 \\ \text{else } s_2 \text{ fi}; s\}, q)\ cl(t) \\ \rightarrow ob(o, C, a, \{l|s_2; s\}, q)\ cl(t) \end{array}}$$

(await2)

$$\frac{\neg[\![g]\!]^{cn}_{(a \circ l), t}}{\begin{array}{c} \{ob(o, C, a, \{l|\text{await } g; s\}, q)\ cl(t)\ cn\} \\ \rightarrow \{ob(o, C, a, \{l|\text{release}; \text{await } g; s\}, q) \\ cl(t)\ cn\} \end{array}}$$

(Read-Fut)

$$\frac{\begin{array}{c} v \neq \bot \quad f = [\![e]\!]^{\varepsilon}_{(a \circ l), t} \\ a(\text{mycomp}) = n \quad r > 0 \end{array}}{\begin{array}{c} ob(o, C, a, \{l|x := e.\text{get}; s\}, q) \\ fut(f, v)\ dc(n, r, u)\ cl(t) \\ \rightarrow ob(o, C, a, \{l|x := v; s\}, q) \\ fut(f, v)\ dc(n, r-1, u)\ cl(t) \end{array}}$$

(New-Object)

$$\frac{\begin{array}{c} \text{fresh}(o') \quad a(\text{mycomp}) = n \quad r > 0 \\ p = \text{init}(B) \quad a' = \text{atts}(B, \overline{v}, o', n) \end{array}}{\begin{array}{c} ob(o, C, a, \{l|x := \text{new } B(\overline{e}); s\}, q) \\ cl(t)\ dc(n, r, u) \\ \rightarrow ob(o, C, a, \{l|x := o'; s\}, q)\ cl(t) \\ ob(o', B, a', p, \emptyset)\ dc(n, r-1, u) \end{array}}$$

(progress)

$$\frac{\text{canAdv}(cn, t)}{\begin{array}{c} \{cn\ cl(t)\} \\ \rightarrow \{\text{Adv}(cn)\ cl(t+1)\} \end{array}}$$

(release)

$$\begin{array}{c} ob(o, C, a, \{l|\text{release}; s\}, q) \\ \rightarrow ob(o, C, a, \text{idle}, \\ \text{enqueue}(\{l|s\}, q)) \end{array}$$

(activate)

$$\frac{p = \text{select}(q, a, cn, t)}{\begin{array}{c} \{ob(o, C, a, \text{idle}, q)\ cl(t)\ cn\} \\ \rightarrow \{ob(o, C, a, p, q \backslash p)\ cl(t)\ cn\} \end{array}}$$

Fig. 4. Semantics for timed Creol

tables. Assume given functions bind$(o, f, m, \overline{v}, C)$ which returns a process resulting from the method activation of m in a class C with actual parameters $\overline{v}$, callee o and associated future f; init(C) which returns a process initializing instances of class C; and atts$(C, \overline{v}, o, n)$ which returns the initial state of an instance of class C with class parameters $\overline{v}$, identity o, and deployment component n. The predicate fresh(n) asserts that a name n is globally unique (where n may be an identifier for an object or a future). Let 'idle' denote any process $\{l|s\}$ where s is an empty statement list. Finally, we define different assignment rules for side effect free expressions (*assign1* and *assign2*), object creation (*new-object*), method calls (*async-call*), and future dereferencing (*read-fut*).

Rule *skip* consumes a **skip** in the active process and a resource in the object's deployment component. Here and in the sequel, the variable s will match any (possibly empty) statement list, the object's deployment component is given by a(mycomp), and $r > 0$ asserts that the deployment component has available resources. Rules *assign1* and *assign2* assign the value of expression e to a variable x in the local variables l or in the fields a, respectively, consuming a resource in the deployment component of the object. Rules *cond1* and *cond2* branch 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. Three operations are used to manipulate a process queue q: enqueue(p, q) adds a process p to q, $q \setminus p$ removes the process p from q, and select(q, a, cn, t) selects a process from q (if q is empty, this is the idle process or no process is *ready* [19]). The actual definitions of these operations are left undefined; different definitions correspond to different scheduling policies for processes. Let $\emptyset$ denote the empty queue. Rule *release* suspends the active process to the process queue, leaving the active process idle. Rule *await1* consumes the await statement if the guard evaluates to true in the current state of the object, rule *await2* adds a release statement in order to suspend the process if the guard evaluates to false. Rule *activate* selects a process from the process queue for execution if this process is *ready* to execute, i.e., if it would not directly be resuspended or block the processor [19].

Communication and Object Creation. Rule *async-call* sends an invocation message to o' with the unique identity f (by the condition fresh(f)) of a new future, the method name m, and actual parameters $\overline{v}$. Note that the return value of the new future f is undefined (i.e., $\perp$). This operation consumes a resource. Rule *bind-mtd* consumes an invocation method and places the process corresponding to the method activation in the process queue of the callee. Note that a reserved local variable 'destiny' is used to store the identity of the future associated with the call. Rule *return* places the return value into the call's associated future. This operation consumes a resource. Rule *read-fut* dereferences the future f in the case where $v \neq \perp$. This operation consumes a resource. Note that if this attribute is $\perp$ the reduction in this object is *blocked*. Finally, *new-object* creates a new object with a unique identifier o'. The object's fields are given default values by atts$(B, \overline{v}, o', n)$, extended with the actual values $\overline{v}$ for the class parameters, o' for this, and n for mycomp. In order to instantiate the remaining attributes, the process p is loaded (we assume that this process reduces to idle if init(B) is 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

$$
\begin{array}{lr}
\text{canAdv}(cn', t) = \mathit{true} & \textit{cn' contains no objects or messages} \\
\text{canAdv}(\mathit{msg}\ cn, t) = \mathit{false} & \textit{messages are instantaneous} \\
\text{canAdv}(\mathit{ob}(o, C, a, p, q)\ \mathit{dc}(n, 0, u)\ cn, t) & \textit{no more resources} \\
\quad = \text{canAdv}(\mathit{dc}(n, 0, u)\ cn, t) \wedge a(\text{mycomp}) == n & \\
\text{canAdv}(\mathit{ob}(o, C, a, \{l|x := f.\text{get}; s\}, q)\ \mathit{fut}(f, \bot)\ cn, t) & \textit{o is blocked and} \\
\quad = \text{canAdv}(\mathit{fut}(f, \bot)\ cn, t) & \textit{no value is available} \\
\text{canAdv}(\mathit{ob}(o, C, a, \text{idle}, q)\ cn, t) & \textit{no ready processes} \\
\quad = \text{canAdv}(cn, t) \wedge \text{select}(q, a, cn, t) == \text{idle} & \\
\text{canAdv}(\mathit{ob}(o, C, a, p, q)\ cn, t) = \mathit{false} & \textit{otherwise}
\end{array}
$$

$$
\begin{array}{lll}
\text{Adv}(\mathit{dc}(n, r, u)\ cn) & = \mathit{dc}(n, u, u)\ \text{Adv}(cn) & \\
\text{Adv}(cn) & = cn & \textit{otherwise}
\end{array}
$$

Fig. 5. Auxiliary functions controlling time advance. Here, *msg* denotes a message and cn' ranges over message- and object-free configurations.

object creation in a named deployment component differs from *new-object* 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 reflect 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. 5), which can be explained as follows:

– For simplicity, we assume that invocation messages do not take time. Therefore, time may *not* advance when a message is on its way.
– Time may *not* advance if any deployment component has remaining resources and any of the component's objects *o* may perform an action. 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 can be activated.
  3. the active process in *o* is not blocked.
– If all deployment components have run out of resources or none of their objects can proceed, then time can advance.

If there can be no activity in any object and no messages are in transit, then time may advance. (A timed model of communication may be obtained by introducing explicit delays in the model, associated with specific method calls, see Sect. 4.) 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. 5, 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 [22].

6 Simulating and Testing the Example

The operational semantics presented in Section 5 can be directly represented in rewriting logic [21], which allows models to be analyzed using the rewrite tool Maude [10]. 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 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 6 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 intervals; `SyncClient` sends an `order` `c` time intervals after the last call returned.

Figure 7 displays the results of two sets of simulation runs over 100 clock cycles. For synchronous clients, 10 to 50 clients and 10 to 50 resources on the `shop` deployment unit were used. Starting with 20 clients, the number of requests goes up linearly with the number of resources, indicating that the system is running at full capacity. Moreover, the number of successful requests decreases somewhat with increasing clients since communication costs also increase. For the periodic case, the system gets overloaded much more quickly since clients will have several pending requests; hence, only 2 to 10 periodic clients were 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 [17]. 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

```
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. 6. Deployment environment and client models of the web shop example

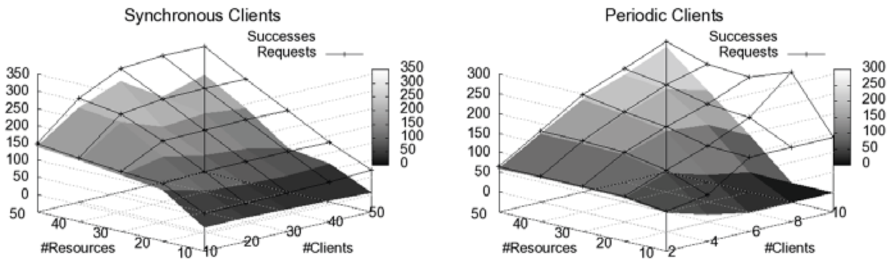


Fig. 7. Number of total and successful requests, depending on the number of clients and resources, for synchronous (left) and periodic (right) clients

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 [16].

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 Fig. 6) but with unlimited resources in the component 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 resources results in a trace $\langle 12, t\rangle$, $\langle 17, t\rangle$, $\langle 22, t\rangle$, ... If the model and system under test have identical untimed behavior, we conclude that a system without resource limits and a deployment component with 50 resources behave equivalently under the assumed workload, whereas deploying with 20 resources 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., [1,2,4,7,11,14,28]). A distinguishing feature of Creol is the cooperative scheduling between asynchronously called methods [19], which allows active and reactive behavior to be combined within objects as well as compositional verification of partial correctness properties [2,11]. Creol's model of cooperative scheduling has recently been generalized to concurrent object groups in Java [26] by restricting to a single activity within the group. Our paper generalizes concurrent object groups to a resource-constrained deployment components, in which the allowed activity per time interval is parametric in concurrent resources, using a time version of Creol [6]. This allows us to abstractly model the effect of deploying concurrent object groups on deployment components with different amounts of processing capacity. A companion paper considers deployment components and resources as first-class citizens of the language [20], which allows load balancing strategies to be modeled in Creol.

Techniques and methodologies for predictions or analysis of non-functional properties are based on either *measurement* and *modeling*. 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 [12]. 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., [8,9,13,15]) have been applied in the embedded software domain, but also to the schedulability of processes in concurrent objects [18]. The latter work complements ours as it does not consider resource restrictions on the concurrency model, but associates deadlines with method calls.

Work on modeling object-oriented systems with resource constraints is more scarce. Using the UML SPT profile for schedulability, performance and time, Petriu and Woodside [23] 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 an operation's set of resources. CSM aims to bridge the gap between UML 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 [27], 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 differ. VDM++ has multi-thread concurrency, preemptive scheduling, and a strict separation of synchronous method calls and asynchronous signals, in contrast to our work with concurrent objects, cooperative scheduling, and caller decided synchronization. In contrast to our fairly succinct semantics, the extension to VDM++ is embedded into VDM++ itself and defined in terms of 100 pages of VDM++ specifications [27].

8 Conclusions and Future Work

This paper proposes a formal model of concurrent processing resources for the deployment of timed object-oriented components. We extend Creol with a notion of deployment component which is parametric in its concurrent resources per time interval and formalize the operational semantics of object execution on deployment components. Based on this formalization, we use the rewriting logic tool Maude to validate resource requirements that are needed to maintain the timed behavior of concurrent objects 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 in a deployment component within 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 the cost of processor swapping and internal control flow. The model may be refined by associating deadlines to method calls and by defining explicit scheduling policies [18]; by computing worst-case costs for the evaluation of expressions [3] and internal control flow; by reconfiguration in terms of object mobility; as well as stronger analysis methods such as, e.g., static analysis and bisimulation techniques.

The abstract notion of resource proposed in this paper reflects computational limitations of concurrent or interleaved activity. Combined with a flexible time model, this 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, approaches such as the one presented in this paper address a need in formal methods, in order to capture models which vary over underlying architectures, for example in software product lines.

References

1. Agha, G.A.: ACTORS: A Model of Concurrent Computations in Distributed Systems. MIT Press, Cambridge (1986)
2. Ahrendt, W., Dylla, M.: A system for compositional verification of asynchronous objects. Science of Computer Programming (2010) (in press)
3. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: COSTA: Design and implementation of a cost and termination analyzer for java bytecode. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2007. LNCS, vol. 5382, pp. 113–132. Springer, Heidelberg (2008)
4. Armstrong, J.: Programming Erlang: Software for a Concurrent World. Pragmatic Bookshelf (2007)
5. Balsamo, S., Marco, A.D., Inverardi, P., Simeoni, M.: Model-based performance prediction in software development: A survey. IEEE Transactions on Software Engineering 30(5), 295–310 (2004)
6. Bjørk, J., Johnsen, E.B., Owe, O., Schlatte, R.: Lightweight time modeling in Timed Creol. In: Electronic Proc. in Theoretical Computer Science, vol. 36, pp. 67–81 (2010); Proc. Intl. Workshop on Rewriting Techniques for Real-Time Systems (RTRTS 2010)
7. Caromel, D., Henrio, L.: A Theory of Distributed Object. Springer, Heidelberg (2005)
8. Chakrabarti, A., de Alfaro, L., Henzinger, T.A., Stoelinga, M.: Resource interfaces. In: Alur, R., Lee, I. (eds.) EMSOFT 2003. LNCS, vol. 2855, pp. 117–133. Springer, Heidelberg (2003)

9. Chen, X., Hsieh, H., Balarin, F.: Verification approach of Metropolis design framework for embedded systems. Intl. J. of Parallel Prog. 34(1), 3–27 (2006)
10. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Quesada, J.F.: Maude: Specification and programming in rewriting logic. Theoretical Computer Science 285, 187–243 (2002)
11. de Boer, F.S., Clarke, D., Johnsen, E.B.: A complete guide to the future. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 316–330. Springer, Heidelberg (2007)
12. Epifani, I., Ghezzi, C., Mirandola, R., Tamburrelli, G.: Model evolution by run-time parameter adaptation. In: Proc. ICSE 2009, pp. 111–121. IEEE, Los Alamitos (2009)
13. Fersman, E., Krcál, P., Pettersson, P., Yi, W.: Task automata: Schedulability, decidability and undecidability. Inf. and Comp. 205(8), 1149–1172 (2007)
14. Haller, P., Odersky, M.: Scala actors: Unifying thread-based and event-based programming. Theoretical Computer Science 410(2-3), 202–220 (2009)
15. Hennessy, M., Riely, J.: Information flow vs. resource access in the asynchronous pi-calculus. ACM TOPLAS 24(5), 566–591 (2002)
16. Hessel, A., Larsen, K.G., Mikucionis, M., Nielsen, B., Pettersson, P., Skou, A.: Testing real-time systems using UPPAAL. In: Hierons, R.M., Bowen, J.P., Harman, M. (eds.) FORTEST. LNCS, vol. 4949, pp. 77–117. Springer, Heidelberg (2008)
17. Hierons, R.M., Bogdanov, K., Bowen, J.P., Cleaveland, R., Derrick, J., Dick, J., Gheorghe, M., Harman, M., Kapoor, K., Krause, P., Lüttgen, G., Simons, A.J.H., Vilkomir, S.A., Woodward, M.R., Zedan, H.: Using formal specifications to support testing. ACM Computing Surveys 41(2) (2009)
18. Jaghoori, M.M., de Boer, F.S., Chothia, T., Sirjani, M.: Schedulability of asynchronous real-time concurrent objects. Journal of Logic and Algebraic Programming 78(5), 402–416 (2009)
19. Johnsen, E.B., Owe, O.: An asynchronous communication model for distributed concurrent objects. Software and Systems Modeling 6(1), 35–58 (2007)
20. Johnsen, E.B., Owe, O., Schlatte, R., Tapia Tarifa, S.L.: Dynamic resource reallocation between deployment components. In: Zhu, H. (ed.) ICFEM 2010. LNCS, vol. 6447, pp. 646–661. Springer, Heidelberg (2010)
21. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science 96, 73–155 (1992)
22. Ölveczky, P.C., Meseguer, J.: Abstraction and completeness for Real-Time Maude. In: Proc. WRLA. ENTCS, vol. 176, pp. 5–27. Elsevier, Amsterdam (2007)
23. Petriu, D.B., Woodside, C.M.: An intermediate metamodel with scenarios and resources for generating performance models from UML designs. Software and System Modeling 6(2), 163–184 (2007)
24. Plotkin, G.D.: A structural approach to operational semantics. Journal of Logic and Algebraic Programming 61, 17–139 (2004)
25. Pohl, K., Böckle, G., Van Der Linden, F.: Software Product Line Engineering: Foundations, Principles, and Techniques. Springer, Heidelberg (2005)
26. Schäfer, J., Poetzsch-Heffter, A.: JCoBox: Generalizing active objects to concurrent components. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 275–299. Springer, Heidelberg (2010)
27. Verhoef, M., Larsen, P.G., Hooman, J.: Modeling and validating distributed embedded real-time systems with VDM++. In: Misra, J., Nipkow, T., Karakostas, G. (eds.) FM 2006. LNCS, vol. 4085, pp. 147–162. Springer, Heidelberg (2006)
28. Welc, A., Jagannathan, S., Hosking, A.: Safe futures for Java. In: Proc. OOPSLA 2005, pp. 439–453. ACM Press, New York (2005)
29. Yacoub, S.M.: Performance analysis of component-based applications. In: Chastek, G.J. (ed.) SPLC 2002. LNCS, vol. 2379, pp. 299–315. Springer, Heidelberg (2002)

Verification of Software Product Lines with Delta-Oriented Slicing

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 line (SPL) engineering is 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. In this paper, we present delta-oriented slicing, an approach to reduce the deductive verification effort across an SPL where individual products are Java programs and their relations are described by deltas. On the specification side, we extend the delta language to deal with formal specifications. On the verification side, we combine proof slicing and similarity-guided proof reuse to ease the verification process.

1 Introduction

A software product line (SPL) [18] 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 SPL [22,23,21]. This has resulted in a modeling approach called delta-oriented programming (Sect. 2). Our current effort aims to exploit the structural 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, Sect. 3). Considering other possibilities to verify SPL that are more meta-level (like generic or partial proofs) and require more semantical information, we decided to go on with a more light-weight approach first.

⋆ This author has been supported by Deutsche Forschungsgemeinschaft (DFG) and by the EU project FP7-ICT-2007-3 HATS.

The technology that we are using to illustrate our approach is Java for programming single products, JML [13] for formal specifications and the KeY system [4] for deductive verification. However, we only make the following assumptions about the verification system:

– We concentrate on systems that manipulate an explicit proof object in the proof assistant style, but do discuss systems operating in the verifying compiler style (a verification condition-generating tool chain with an SMT solver at its end).
– We support both ways in which 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 parametric on how a verification system treats invariants. In the worst case, all methods in the program have 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 requiring that all *relevant* invariants must be checked.

In our approach we analyze the SPL model to determine which parts of the original product are unchanged in the new product and also do not have to be verified again. This analysis constitutes proof slicing (Sect. 4).

For the modified or otherwise affected product parts, we apply a previously-developed proof reuse technique based on the assumption of similarity between the two implementation variants. (Sect. 5).

We present related work in Sect. 6 and draw conclusions in Sect. 7.

2 Delta-Oriented Programming of Software Product Lines

Delta-oriented programming (DOP) [22,23,21] is a novel approach for implementing software product lines. Delta-oriented programming offers an expressive and flexible "programming meta-language" for specifying a set of products. Its aim is to relax the restrictions of currently established SPL description formalisms such as feature-oriented programming (FOP) [3] by adding the explicit possibility to remove parts of a program. For a more detailed comparison between delta-oriented and feature-oriented programming, the reader is referred to [22].

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 implementation for some valid feature configuration, which can be developed by conventional single-application engineering techniques. Delta modules specify changes to be applied to the core module 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 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.*[1] *Vice versa, we identify the method signature m with a set of classes in a product that declare a method with that signature: $C \in m$ if C::$m \in C$.*

```
core Base {
    class Account extends Object {
        int balance;
        int bonus;
        void addBonus(int x){}
        void update(int x) {
            balance += x;
        }
    }
}
```

(a) Core module with `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) Result of delta module application

Fig. 1. Example of a delta-oriented product line

Modification operations used in delta modules that we consider in this paper are the following:

- adding/removing a class declaration C: $adds(C)$, $removes(C)$
- modifying class C by

[1] For simplicity, we assume the absence of method overloading. In Java, a class may contain several method implementations with the same identifier and compatible parameter types. This renders the lookup procedure far more complicated; c.f. [8, Sect. 15.12.2].

- adding/removing a field f: $\mathit{adds}(C{::}f)$, $\mathit{removes}(C{::}f)$
- adding/removing a method declaration m: $\mathit{adds}(C{::}m)$, $\mathit{removes}(C{::}m)$
- changing the direct superclass of C to C': $\mathit{reparents}(C, C')$

On an abstract level, 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.[2] The original delta language proposal [22] 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 in this paper.

Example 1. Our running example in this paper is a delta-oriented product line of bank accounts inspired by [7]. Figure 1a shows the core module of this SPL with the basic Account class. Figure 1b shows the delta module `DInvestment` for 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, in Example 3, we will also see the `Paycheck` feature adding the class `Employer` as a client of `Account`. $\Diamond$

3 Delta-Oriented Formal Specification of Software Product Lines

We use the Java Modeling Language (JML) [13] 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 Sect. 2. A core module is specified just as a conventional program. An example of a core module with JML specifications can be seen in the first listing of Example 3.

For delta modules, we extend the delta language with the following operations to manipulate specifications:

– adding an invariant to a class: $\mathit{adds}(C, I)$
– removing an invariant from a class: $\mathit{removes}(C, I)$

[2] This is a slight generalization of the original delta approach, where deltas could only be applied to the core product.

– 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.[3] ◊

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.

```
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 a method

4 Delta-Oriented Slicing

When a new product is derived by delta application, in general, both the implementation as well as the specification change. However, from the structural information available in the used delta modules, we are able to conservatively infer which specifications of the new product remain valid (i.e., the proofs done for the old product are not affected by the change) and which parts have to be (re-)proven in order to establish the specified properties. We call the latter *delta-oriented slice.*

[3] It is possible to specify code and specification changes in the same delta module. The separation at this point is for presentation reasons.

Slicing originated as a program analysis technique answering the question of which program statements influence the value of a given variable. Our algorithm answers the question of which proofs are influenced by a delta module.

Of course, the simplest and safest way to achieve assurance for a changed product is to redo all proofs. However, at the current state of hardware and deduction technology, this approach is too slow for any product of non-trivial size. Our approach is much less computationally expensive as it only involves a deterministic static analysis of different artifacts. This way, proof slicing can quickly provide feedback to the engineer on what impact a certain change to the product will have.

Proof Modularity

The key to obtaining a sound slicing algorithm is identifying non-modular proof steps. The issue of proof non-modularity arises if the validity of certain proof steps in a verification proof is lost when the program that is to be verified is changed or extended.

The change may be explicit, i.e., concerning the source code of the very method being verified, or implicit, i.e., concerning program entities that are only referenced (e.g., other methods called). Explicit changes are easy to detect, and if they are benign, they can be treated by proof reuse (Sect. 5). Implicit changes are more involved, and their impact depends both on the semantics of the programming language, as well as on the particular verification calculus. Implicit change is the case that we concentrate on in the following.

Proof modularity has been recognized as an issue for quite some time, focusing, naturally, on adding/removing classes and overriding methods. Particularly relevant to our effort are a previous account for the KeY system [20, Sect. 6.2] as well as a comprehensive survey for the KIV system [24, Chap. 6]. As our change vocabulary is larger, we have to address this issue anew. In the KeY system, identifying rules resulting in non-modular proof steps is made easier by the fact that the class declarations and the class hierarchy are not part of the original proof obligation. This information is available in the background (i.e., in the prover implementation) and can be introduced into a logical sequent by rules containing *metaconstructs* (functions that are not logically specified, but programmed in the prover). These functions make non-modular rules easily identifiable syntactically, which we have done for the KeY rule base. In the KeY calculus, we discern rules giving rise to proof steps whose validity is:

(A) not affected by implicit program changes (rewriting, propositional, and the like, but also many symbolic execution rules, e.g., for conditionals, loops, etc.);
(B) affected by presence or absence of classes regardless of their content;[4]

[4] The rules of this type are rare and the KeY system has only two of them: TYPEABSTRACT and ARRAYSTORESTATICANALYSE. The former allows deducing the dynamic type of an object pointed to by an expression with an abstract static type (this rule produces a disjunction over all subclasses). The latter uses a simple static analysis to check whether an array assignment can throw an `ArrayStoreException`.

(C) affected by methods declared in classes; these rules are the non-modular method invocation rules inlining the method implementations and simulating dynamic binding;
(D) affected by fields declared in classes regardless whether these fields are used in the program; these are the instance creation rules assigning default values to fields;
(E) affected by inheritance relationship between classes; these are the rules for tackling the inheritance predicate $\sqsubseteq$.

The slicing algorithm is based on these findings.

Other systems encode the class hierarchy as axioms that are part of the proof obligation from the start. Here, it is necessary to analyze the proofs constructed by the prover for occurrence of particular axioms. This may be difficult if there is no explicit proof object, but, for instance, the popular SMT prover Z3 often used in verifying compilers provides this information.

The Algorithm

In the following, we present the delta-oriented slicing algorithm. As the first step of the algorithm, we copy all finished proofs from product P_1 into product P_2 regardless of their validity for P_2. In the resulting 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.

Input: A set of proofs for a product P_1, and the delta $\Delta(P_1, P_2)$[5]
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 $\Delta(P_1, P_2)$. 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.

2. For each $\mathit{adds}(C)$:
   (a) do $\mathit{adds}(C{::}f)$ for each $f \in C$
   (b) do $\mathit{adds}(C{::}m)$ for each $m \in C$
   (c) invalidate all proofs with proof steps by non-modular rules of type (B) where C or any of its superclasses appear in the rule conclusion

[5] For the sake of the algorithm, we assume that $\Delta(P_1, P_2)$ contains exactly one delta module (i.e., we assume delta module composition).

3. For each *removes*(C):
   (a) do *removes*$(C{::}f)$ for each $f \in C$
   (b) do *removes*$(C{::}m)$ for each $m \in C$
   (c) invalidate all proofs with proof steps by non-modular rules of type (B) where C or any of its superclasses appear in the rule conclusion

Adding and removing methods. When adding methods, we have to distinguish if their invocation is treated by inlining and contract application. If an altered implementation is inlined, the proof, of course, will be invalidated. For a contract, this is different since the altered implementation is expected to fulfill the old contract. Contracts are also not affected by method removal. Even though an implementation has been removed, the contract still applies to some overriding implementation in a subclass.

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 Fig. 3)
   (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 (Fig. 3)
   (b) proofs using the contracts for m remain valid

Adding and removing fields. In steps 6–7, it might not be immediately clear why adding or removing a field can invalidate a proof. Consider the following code snippet:

```
class A { Object f; }
class B extends A {  /*@ invariant f == ((A)this).f; @*/ }
```

The invariant in class `B` holds if and only if no field `f` is added to class `B`. Otherwise, the left occurrence of `f` would refer to `B::f`, while the right one would continue referring to `A::f` as fields are bound statically in Java.

Adding or removing fields also invalidates proofs containing instance creation, as this process must assign all fields a default value, resulting in varying intermediate states.

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
   (e) invalidate all pre-existing proofs with proof steps assigning default values (during instance creation) to fields of an object with type $A \sqsubseteq C$

7. For each $\mathit{removes}(C{::}f)$: same as step 6, but look for $C{::}f$ in P_1

Class reparenting. Reparenting is an invasive operation, which is illustrated in Fig. 4. $\mathit{reparents}(C, C')$ moves C from under its old direct supertype $\widetilde{C}$ and beneath C', and with it $\mathit{movedPart} = \{K \mid K \sqsubseteq C\}$. As $\widehat{C}$ we then denote the least common supertype of $\widetilde{C}$ and C'.

Reparenting class C makes C and its subclasses lose features (implementations and specifications) inherited from $\mathit{oldBranch} = \{K \mid \widetilde{C} \sqsubseteq K \sqsubset \widehat{C}\}$ and inherit new features from $\mathit{newBranch} = \{K \mid C' \sqsubseteq K \sqsubset \widehat{C}\}$.

8. For each $\mathit{reparents}(C, C')$:
   (a) invalidate all pre-existing proofs inlining method bodies for any virtual method call e.m`()` with S as the static type of e and
      i. $S \in \mathit{newBranch}$
      ii. $\widehat{C} \sqsubseteq S$

      or, if at least one method body $K{::}m$ was inlined such that
      iii. $S \in \mathit{movedPart}$ and $K \in \mathit{oldBranch}$
      iv. $S \in \mathit{oldBranch}$ and $K \in \mathit{movedPart}$

      This step reacts to changes in the big case distinction simulating dynamic binding.
   (b) invalidate all pre-existing proofs about/inlining any method implementation $C{::}m$ containing a method call of the form `super.`m'`()` (as the superclass will change)
   (c) invalidate all pre-existing proofs about/inlining any method implementation $K{::}m$, $K \in \mathit{movedPart}$ that references a field $K'{::}f$ declared in *oldPart* (as this reference would change its meaning after the move)
   (d) contracts for methods in reparented classes remain valid unless the contract no longer exists (i.e., it was inherited from *oldBranch*)
   (e) invalidate proofs for specifications inherited from any class in *oldBranch*
   (f) prove that all classes $K \in \mathit{movedPart}$ satisfy the specifications inherited from new superclasses in *newBranch*
   (g) invalidate all proofs containing a proof step deciding the predicate $A \sqsubseteq B$ if $A \sqsubseteq C$ and $B \in \mathit{oldBranch}$

Adding and removing specifications.

9. For each $\mathit{adds}(C{::}m, \mathit{ct})$
   (a) prove that the contract ct is fulfilled by all $C'{::}m$ with $C' \sqsubseteq C$

10. For each $\mathit{removes}(C{::}m, \mathit{ct})$
   (a) invalidate all pre-existing proofs that use the contract ct

11. For each $\mathit{adds}(C, I)$
   (a) prove that the invariant I is fulfilled by all relevant implementations

12. For each $\mathit{removes}(C, I)$
   (a) invalidate all pre-existing proofs that assume the invariant I

For some of the algorithm steps, we need to determine whether an implementation $C::m$ is potentially referenced by the method invocation expression `e.m()`.
We consider the three different method invocation modes available in Java, defining for each mode 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 `e` (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.
- The implementation is in S or one of its subclasses: If $C \sqsubseteq S$, then "yes"
- The implementation is in a superclass of S, but it is inherited by S or one of its subclasses (i.e., it is not overridden between C and S): If $S \sqsubseteq C$ such that for all K with $S \sqsubseteq K \sqsubset C$ holds $K \notin m$, then "yes" (cf. Fig. 5).
- 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 `super.m()`).
- If $S \sqsubseteq C$ and for all K with $S \sqsubseteq K \sqsubset C$ holds $K \notin m$, then "yes".
- Otherwise, "no".

Fig. 3. Subroutine: When is a method implementation potentially referenced?

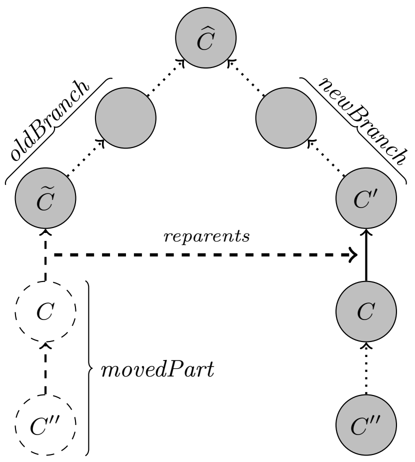


Fig. 4. Illustration of *reparents*(C, C'). Solid lines represent the direct subtype relation, dotted lines its transitive closure, and dashed lines show relations of the previous product.

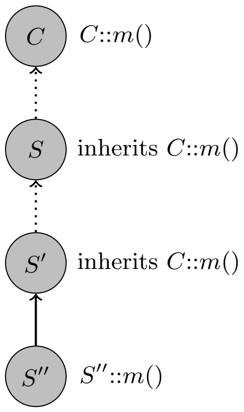


Fig. 5. Virtual method invocation mode and method overriding

An Example

Example 3. (i) We return to the bank account example introduced in Sect. 2. The core product with the basic `Account` class now contains specifications (see 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 step 4 for the added method, step 6 for the added field and step 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

```
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);
        }
    }
}
```

empty) as the field `a` did not exist previously. Step 9 is subsumed by step 4 as `Employer` has no subclasses. No proofs are invalidated.

(iii) If we now want to incorporate the `Investment` feature as well, we apply the deltas `DInvestment` (Fig. 1b) and `DInvestmentSpec` (Fig. 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: step 4 for the added methods, step 5 for the removed methods, step 9 for the added contract and step 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 feasible here (see Sect. 5). 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). $\diamondsuit$

5 Proof Reuse for Changed Methods

In this section, we point to the existing technique of proof reuse [11] as a natural complement to delta-oriented proof slicing. This part of our approach is tailored to interactive verification systems like KeY, where the user provides hints to the prover by manipulating an explicit proof object. In practice (although not in our illustrating example), proofs contain proof steps which 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 through proof reuse.

The proof reuse technique has been originally developed for KeY by one of the authors 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. 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. It is a light-weight technique based on proof replay rather than on proof generation. For a full account of proof reuse in KeY we refer the reader to [11].

In the delta-oriented slicing step, we have identified which proofs have to be redone for the newly generated product. However, some of the changed method bodies may still have considerable similarities to the ones in the already verified product. The correctness proofs of such modified methods are likely to resemble the old proofs. Here proof reuse can help. Reuse can also be used in case of 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 [1] or product models [14]. Further approaches (e.g., [6]) verify that the variability specified by a feature model is correctly implemented in code. Efficient verification of product behavior, however, is not well established. In testing [15,17] or model checking [12,5] there is work to make validation of product lines more efficient, though.

In [2], 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 in line 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.

Reuse of verification artifacts is also related to a whole plethora of work which is impossible to survey here, such as slicing for debugging [25,27] or model checking [9], reuse of refined specifications [26], change management in theory development [16,10], incremental compilation, refactoring, and software change impact analysis.

An interesting and closely related result from change impact analysis is the tool Chianti [19], which determines whether the results of a test are affected by changes to the source code. Changes to the program are decomposed into "atomic operations", which are similar to our delta operations. These are then analyzed for their impact on the program's call graph.

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 identified several interesting lines of future research. Most of them regard the transition from a syntactic modeling of SPL as in the current delta-oriented programming approach [22] 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. Another reason 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 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.

Finally, getting the formal specification of a product right is difficult, but deriving a correct product from another also has its pitfalls. 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. The product derivation process may seduce one to believe that I is still an adequate specification for the new product, which might not be the case. In the simplest case I might contain pure methods, which have changed between products. The issue is aggravated by the complicated and sometimes unclear semantics of modern specification languages and requires further investigation.

References

1. Batory, D.S., Benavides, D., Ruiz-Cortés, A.: Automated analysis of feature models: Challenges ahead. Communications of the ACM 49(12) (2006)
2. Batory, D.S., Börger, E.: Modularizing theorems for software product lines: The Jbook case study. Journal of Universal Computer Science 14(12) (2008)
3. Batory, D.S., Sarvela, J.N., Rauschmayer, A.: Scaling step-wise refinement. IEEE Trans. Software Eng. 30(6), 355–371 (2004)
4. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software. LNCS (LNAI), vol. 4334. Springer, Heidelberg (2007)
5. Classen, A., Heymans, P., Schobbens, P.-Y., Legay, A., Raskin, J.-F.: Model checking lots of systems: Efficient verification of temporal properties in software product lines. In: 32nd International Conference on Software Engineering, ICSE 2010, Cape Town, South Africa, May 2-8. IEEE, Los Alamitos (2010) (to appear)
6. Czarnecki, K., Pietroszek, K.: Verifying feature-based model templates against well-formedness OCL constraints. In: Conf. on Generative Programming and Component Engineering (GPCE) (2006)
7. Delaware, B., Cook, W., Batory, D.: A Machine-Checked Model of Safe Composition. In: Foundations of Aspect-Oriented Languages (FOAL), pp. 31–35. ACM, New York (2009)
8. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java Language Specification, 3rd edn. Addison-Wesley Longman, Amsterdam (2005)
9. Hatcliff, J., Dwyer, M.B., Zheng, H.: Slicing software for model construction. Higher-Order and Symbolic Computation 13(4), 315–353 (2000)
10. Hutter, D.: Management of change in structured verification. In: Automated Software Engineering (ASE), p. 23 (2000)
11. Klebanov, V.: Proof reuse. In: Beckert et al. [4]
12. Lauenroth, K., Pohl, K., Toehning, S.: Model checking of domain artifacts in product line engineering. In: Automated Software Engineering (ASE), pp. 269–280. IEEE Computer Society, Los Alamitos (2009)

13. 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)
14. Mannion, M.: Using First-Order Logic for Product Line Model Validation. In: Chastek, G.J. (ed.) SPLC 2002. LNCS, vol. 2379, pp. 176–187. Springer, Heidelberg (2002)
15. McGregor, J.D.: Testing a software product line. Technical Report CMU/SEI-2001-TR-022, Software Engineering Institute, Carnegie Mellon University (December 2001)
16. Mossakowski, T.: Heterogeneous theories and the heterogeneous tool set. In: Kalfoglou, Y., Schorlemmer, W.M., Sheth, A.P., Staab, S., Uschold, M. (eds.) Semantic Interoperability and Integration. Dagstuhl Seminar Proceedings, vol. 04391, IBFI, Schloss Dagstuhl (2005)
17. Muccini, H., van der Hoek, A.: Towards testing product line architectures. Electr. Notes Theor. Comput. Sci 82(6) (2003)
18. Pohl, K., Böckle, G., van der Linden, F.: Software Product Line Engineering: Foundations, Principles, and Techniques. Springer, Heidelberg (2005)
19. Ren, X., Shah, F., Tip, F., Ryder, B.G., Chesley, O.: Chianti: A tool for change impact analysis of Java programs. In: Vlissides, J.M., Schmidt, D.C. (eds.) Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2004, Vancouver, BC, Canada, October 24-28, pp. 432–448. ACM, New York (2004)
20. Roth, A.: Specification and Verification of Object-oriented Software Components. PhD thesis, Universität Karlsruhe (2006)
21. Schaefer, I.: 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)
22. Schaefer, I., Bettini, L., Bono, V., Damiani, F., Tanzarella, N.: Delta-Oriented Programming of Software Product Lines. In: Bosch, J., Lee, J. (eds.) SPLC 2010. LNCS, vol. 6287, pp. 77–91. Springer, Heidelberg (2010)
23. Schaefer, I., Worret, A., Poetzsch-Heffter, A.: A model-based framework for automated product derivation. In: Model-driven Approaches in Software Product Line Engineering (MAPLE 2009) (2009)
24. Stenzel, K.: Verification of Java Card Programs. PhD thesis, Fakultät fur angewandte Informatik, University of Augsburg (2005)
25. Tip, F.: A survey of program slicing techniques. Journal of Programming Languages 3(3) (1995)
26. Wehrheim, H.: Slicing techniques for verification re-use. Theor. Comput. Sci. 343(3), 509–528 (2005)
27. Weiser, M.: Program slicing. IEEE Transactions on Software Engineering 10(4), 352–357 (1984)

Satisfiability Solving and Model Generation for Quantified First-Order Logic Formulas

Christoph D. Gladisch

Karlsruhe Institute of Technology (KIT)
Institute for Theoretical Informatics
Germany
gladisch@uni-koblenz.de

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., [10,23,13,24]). 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. Consequently, the technique is not a decision procedure, i.e. it may not terminate. However, it can solve more general formulas than SMT solvers can solve in cases where it terminates. As a motivating example, assume we want to show the satisfiability of the formula ϕ[1] which we define as

$$\forall x.(x \geqslant 0 \rightarrow \mathit{prev}(\mathit{next}(x)) = x) \tag{1}$$

where *prev* and *next* are uninterpreted function symbols. Some state-of-the-art SMT solvers — concretely we have tested Z3 [9], CVC3 [1], Yices [12] — 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.

While model generation is not a new idea, the novelty of our approach are (1) the choice of language to syntactically 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.

We experience that in software verification much time is spend with fixing and adjusting the programs, specifications, and annotations. For instance, Figure 2 shows an unprovable verification condition with subformulas similar to formula ϕ[1]. It is invaluable to detect if such verification conditions have counter examples. Once a program is correct and annotations are strong enough a verification tool can afterwards prove the correctness of the program usually automatically. The generation of counter examples is further important in counter example guided abstraction refinement (CEGAR) [7] and for checking the consistency, i.e. contradiction-freeness, of axiomatizations and of preconditions in specifications.

This paper is based on the technical report [17] where we propose our approach for the first time. In [18], which is a paper following this paper, we describe an algorithm that implements our approach. While in [18] we describe the application of the algorithm to test data generation and its evaluation, the contributions of this paper are the description of the theory of our approach and a soundness proof.

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 [23] which was first used in the theorem prover Simplify [11]. 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 [21]) 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 [22]. 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 [22]). 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 [14]. An important fragment of FOL for program specification which allows a complete instantiation is the Array Property Fragment [6]. E-matching is used in state-of-the-art SMT solvers such as Z3 [9], CVC3 [1], Yices [12], and others (see [8]). Formula ϕ_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. [16]). These approaches focus, however, on validity or respectively unsatisfiability proofs and not on satisfiability solving. More precisely, they do not guarantee a complete instantiation of quantifiers in general case.

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 [27] 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.

Model generation theorem provers (MGTP) are similar to SMT solvers as their underlying technique is DPLL lifted to FOL. For instance, the theorem prover Darwin [2] is an instantiation-based prover which is sound and complete for the unsatisfiability of FOL, i.e. without theories. For the satisfiability part it decides Bernays-Schönfinkel formulas.

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., [15] for more examples). These techniques are, in contrast to the proposed technique, not capable of eliminating the quantifier in, e.g., ϕ_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 approach is, however, that it is not restricted to a certain class of formulas.

Quantified constraint satisfaction problem (QCSP) solvers primarily regard the finite version of the satisfiability problem, whereas our approach handles infinite domains. Some of the work, e.g. [4], also considers continues domains, however, these techniques do not handle uninterpreted function symbols other than constants.

Finite and infinite model building techniques are described in [25]. The authors distinguish between enumeration-based methods corresponding to the above mentioned instantiation techniques and deduction-base methods which are in the main focus of the book. Deductive methods produce syntactic representations of models in some logical language. Nitpik which is the counter example generator of Isabel/HOL uses first-order relational logic (FORL) [5]. FORL extends FOL with relational calculus operators and the transitive closure. The approach we propose is a deduction-based method which differs from existing approaches in the representation and generation of models.

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, i.e. a partial interpretation, 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 concept. 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 ϕ_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 ϕ_1 with respect to this program, i.e. $wp((4), \phi_1)$. 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 reducing the chance of introducing new quantified formulas.

3 KeY's Dynamic Logic with Updates

The KeY system [3,20] 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) [19]. 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* [26].

Notation. We use the following abbreviations for syntactic entities: V is the set of (logic) variables; Σ^f is the set of function symbols; $\Sigma^f_r \subset \Sigma^f$ is the set of rigid function symbols, i.e. functions with a fixed interpretation such as, e.g., '0', 'succ', '+'; $\Sigma^f_{nr} \subset \Sigma^f$ is the set of non-rigid function symbols, i.e. uninterpreted functions; Σ^p is the set of predicate symbols; Σ is the signature consisting of $\Sigma^f \cup \Sigma^p$; Trm_{FOL} is the set of FOL terms; Trm is the set of DL terms; Fml_{FOL} is the set of FOL formulas; Fml is the set of DL formulas; U is the set of updates; $\doteq$ is the equality predicate; and $=$ is syntactic equivalence. The following abbreviations describe semantic sets: $\mathcal{D}$ is the FOL domain or universe; $\mathcal{S}$ is the set of states or equivalently the set of FOL interpretations. To describe semantic properties we use the following abbreviations: $val_s(t) \in \mathcal{D}$ is the valuation of $t \in Trm$ and $val_s(u) \in \mathcal{S}$ is the valuation of $u \in U$ in $s \in \mathcal{S}$; $s \vDash \varphi$ means that φ is true in state $s \in \mathcal{S}$; $\vDash \varphi$ 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 is a modality which moves the interpretation to a new Kripke state and we say an update *changes* an interpretation. The states differ in the interpretation of symbols Σ^f_{nr} such as uninterpreted functions. 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 $(\{a := b\}a \doteq c) \in Fml$, 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 $a \doteq c$. The meaning of this *update application* is the same as that of the weakest precondition $wp(a := b, a \doteq c)$, i.e. it represents all states such that after the assignment $a := b$ the formula $a \doteq c$ is true — which is equivalent to $b \doteq c$.

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 $u \in U$ with no free variables 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 $u \in U$ with no free variables 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, if not stated otherwise.*

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 $\delta_1, \ldots, \delta_m \in \Delta$ are closed formulas. Sequents are normally, e.g. in [3], 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 a free occurrence of x.*

Terms and Formulas

- *$val_s(\{u\}t) \equiv val_{s'}(t)$, where $s' \equiv val_s(u)$*
- *$val_s(\{u\}\varphi) \equiv val_{s'}(\varphi)$, where $s' \equiv val_s(u)$*

Updates

- *$val_s(\varepsilon) \equiv s$*
- *$val_s(f(t_1,\ldots,t_n) := t) \equiv s'$, where s' is the same as s except the interpretation of f is changed such that $val_{s'}(f(t_1,\ldots,t_n)) \equiv val_s(t)$.*
- *$val_s(u_1;u_2) \equiv s'$, there is s'' with $s'' \equiv val_s(u_1)$ and $s' \equiv val_{s''}(u_2)$*
- *$val_s(u_1 \,||\, u_2) \equiv s'$. We define s' by the interpretation of terms t. Let $v_0 \equiv val_s(t)$, $v_1 \equiv val_s(\{u_1\}t)$, and $v_2 \equiv val_s(\{u_2\}t)$.*

$$\text{If } v_0 \not\equiv v_2 \text{ then } val_{s'}(t) \equiv v_2 \text{ else } val_{s'}(t) \equiv v_1.$$

- *$val_s(\texttt{if}\ \ \varphi;\, u) \equiv s'$, if $val_s(\varphi) \equiv true$ then $s' \equiv val_s(u)$, otherwise $s' \equiv s$.*
- *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 a well-ordering $\succ$ 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 [26,3] 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.

- $\{f(1) := a\}f(2) \doteq f(1)$ simplifies to $f(2) \doteq a$.
- $\{f(b) := a\}P(f(c))$ 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.
- $\{u; f(t_1,\ldots,t_n) := t\}$ is equivalent to $\{u \,||\, f(\{u\}t_1,\ldots,\{u\}t_n) := \{u\}t\}$.
- $\{f(1) := a \,||\, f(2) := b\}f(2) \doteq f(1)$ simplifies to $b \doteq a$.
- $\{f(1) := a \,||\, f(1) := b\}f(2) \doteq f(1)$ simplifies to $f(2) \doteq b$, 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 set of models 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 consider the model generation problem of a quantified formula $\forall x.\phi(x)$ in a sequent $\varphi = (\Gamma, \forall x.\phi(x) \implies \Delta)$. 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:*

$$(\{u\}(\Gamma, \forall x.\phi(x) \Longrightarrow \Delta)) \equiv (\{u\}(\Gamma, true \Longrightarrow \Delta)) \tag{5}$$

If this problem is solved by a technique, then this technique can be applied iteratively to all quantified formulas occurring in Γ and Δ resulting in a sequent $\Gamma' \Longrightarrow \Delta'$ that consists only of ground formulas. Note that non-skolemizable quantified formulas occurring 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* $\models \{u\}\forall x.\phi(x)$*, without analysing the semantic properties of the matrix* $\phi(x)$*.*

b) The theorem prover defined in [3] is not sufficiently powerful to simplify $(\Gamma, \{u\}\forall x.\phi(x) \Longrightarrow \Delta)$ *to* $(\Gamma, true \Longrightarrow \Delta)$ *if* $\models \{u\}\forall x.\phi(x)$ *and* u *is a quantified update.*

Some possibilities to analyse the semantic properties of $\phi(x)$ are to test instances of $\phi(x)$ or to use free variables (see, e.g., [16]). We have experimented with the latter approach and could solve problem (a) in several cases but we describe a better approach in this paper. 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 is described and evaluated in [18] but without a soundness proof. In this section we provide a theorem that formalizes only the crucial problem simplification technique of the algorithm and prove it.

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* $\phi \in \Sigma^p$*. Let* $u_0, \ldots, u_m \in U$*;* $\alpha_0, \ldots, \alpha_m \in Fml$*, with* $m \in \mathbb{N}$*, be closed by quantifiers. The formulas* $\psi_m, \varphi'_m, \varphi_m \in Fml$*, are defined recursively as:*

- $\varphi_0 = (\Gamma, \underline{\forall x.\phi(x)} \Longrightarrow \Delta)$ $\qquad \varphi_{m+1} = \{u_m\}(\alpha_m \to \varphi_m)$
- $\varphi'_0 = (\Gamma, \underline{true} \Longrightarrow \Delta)$ $\qquad \varphi'_{m+1} = \{u_m\}(\alpha_m \to \varphi'_m)$
- $\psi_0 = (\Gamma \Longrightarrow \underline{\forall x.\phi(x)}, \Delta)$ $\qquad \psi_{m+1} = \{u_m\}(\alpha_m \to \psi_m)$

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* $\varphi = (\Gamma, \gamma \Longrightarrow \Delta)$ *and* $\psi_m, \varphi'_m, \varphi_m \in Fml$*, with* $m \in \mathbb{N}$*, 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 choosing 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 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 of models for ground formulas by using the more general Def. 5 instead of Def. 4 in Theorem 1.

Definition 5. *Given a sequent $\varphi = (\Gamma, \gamma \Rightarrow \Delta)$, where $\Gamma, \Delta \subset Fml$ and $\gamma \in Fml$. Let $u_0, \ldots, u_m \in U$; $\alpha_0, \ldots, \alpha_m \in Fml$, with $m \in \mathbb{N}$, be closed by quantifiers. The formulas $\psi_m, \varphi'_m, \varphi_m \in Fml$ are defined recursively as follows:*

- $\psi_0 = (\Gamma \Rightarrow \gamma, \Delta)$ $\quad\quad$ $\psi_{m+1} = \{u_m\}(\alpha_m \rightarrow \psi_m)$
- $\varphi'_0 = (\Gamma, true \Rightarrow \Delta)$ $\quad\quad$ $\varphi'_{m+1} = \{u_m\}(\alpha_m \rightarrow \varphi'_m)$
- $\varphi_0 = (\Gamma, \gamma \Rightarrow \Delta)$ $\quad\quad$ $\varphi_{m+1} = \{u_m\}(\alpha_m \rightarrow \varphi_m)$

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

$$\underbrace{(\Gamma \Rightarrow \forall x.\phi(x), \Delta)}_{\psi_0} \leftrightarrow (\underbrace{(\Gamma, true \Rightarrow \Delta)}_{\varphi'_0} \leftrightarrow \underbrace{(\Gamma, \forall x.\phi(x) \Rightarrow \Delta)}_{\varphi_0})$$

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.

$$((\Gamma \wedge true) \rightarrow \Delta) \leftrightarrow ((\Gamma \wedge \forall x.\phi(x)) \rightarrow \Delta)$$
$$(\Gamma \rightarrow \Delta) \leftrightarrow ((\Gamma \wedge \forall x.\phi(x)) \rightarrow \Delta)$$

$$\begin{array}{cc}
(\Gamma \rightarrow \Delta) \rightarrow ((\Gamma \wedge \forall x.\phi(x)) \rightarrow \Delta) & ((\Gamma \wedge \forall x.\phi(x)) \rightarrow \Delta) \rightarrow (\Gamma \rightarrow \Delta) \\
((\Gamma \rightarrow \Delta) \wedge \Gamma \wedge \forall x.\phi(x)) \rightarrow \Delta & (((\Gamma \wedge \forall x.\phi(x)) \rightarrow \Delta) \wedge \Gamma) \rightarrow \Delta \\
(\Delta \wedge \Gamma \wedge \forall x.\phi(x)) \rightarrow \Delta & ((\forall x.\phi(x) \rightarrow \Delta) \wedge \Gamma) \rightarrow \Delta \\
(\Delta \wedge \Gamma) \rightarrow \Delta & ((\forall x.\phi(x) \rightarrow \Delta) \wedge \Gamma) \rightarrow \Delta \\
\Delta \rightarrow \Delta & ((\neg\forall x.\phi(x) \wedge \Gamma) \rightarrow \Delta) \wedge ((\Delta \wedge \Gamma) \rightarrow \Delta) \\
true & (\neg\forall x.\phi(x) \wedge \Gamma) \rightarrow \Delta \\
 & \Gamma \rightarrow (\forall x.\phi(x) \vee \Delta)
\end{array}$$

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)

$$\vDash ((\alpha_m \to \psi_m) \leftrightarrow ((\alpha_m \to \varphi'_m) \leftrightarrow (\alpha_m \to \varphi_m))) \tag{7}$$

Due to Lemma 1, (7) implies

$$\vDash \{u_m\}((\alpha_m \to \psi_m) \leftrightarrow ((\alpha_m \to \varphi'_m) \leftrightarrow (\alpha_m \to \varphi_m))) \tag{8}$$

that can be simplified by to (for all operators $\circ$: $\{u\}(A \circ B) \equiv \{u\}A \circ \{u\}B$)

$$\vDash (\{u_m\}(\alpha_m \to \psi_m) \leftrightarrow (\{u_m\}(\alpha_m \to \varphi'_m) \leftrightarrow \{u_m\}(\alpha_m \to \varphi_m))) \tag{9}$$

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 $\varphi^{\neg}_m \in Fml$ that can be recursively defined as

$$\varphi^{\neg}_0 = \neg(\Gamma, true \Longrightarrow \Delta) \qquad \varphi^{\neg}_{m+1} = \{u_m\}(\alpha_m \wedge \varphi^{\neg}_m)$$

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 updates with axioms. 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

- $\vDash \{u\}(\alpha \to \psi)$*, and*
- *there is some $s \in \mathcal{S}$ with $s \vDash \neg\{u\}(\alpha \to \varphi')$*

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 ϕ_{10}:

$$\forall x. x > a \rightarrow f(x) = g(x) + x \tag{10}$$

is satisfiable in any state after execution of the quantified update u_{11}:

$$\texttt{for } x;\ x > a;\ f(x) := g(x) + x \tag{11}$$

i.e. $\vDash \{\phi_{10}\}u_{11}$. Notice the similar syntactical structure between ϕ_{10} and u_{11}. Another solution is u_{12}:

$$\texttt{for } x;\ x > a;\ g(x) := f(x) - x \tag{12}$$

for which $\models \{u_{12}\}\phi_{10}$ holds. 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)

$$(\texttt{for}\ x;\ f(x) \geqslant x;\ g(x) := f(x) + \mathit{gtZero}\ ,\ \mathit{gtZero} > 0)$$
$$(\texttt{for}\ x;\ \neg(g(x) < f(x));\ f(x) := x - \mathit{gtZero}\ ,\ \mathit{gtZero} > 0)$$

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 use backtracking when new quantified formulas are introduced. A limitation is the handling of recursively defined functions. For instance, the following update application, as well as any other, does not eliminate the quantified subformula of the following formula:

$$\{\texttt{for}\ x;\ x > 0;\ h(x) := h(x-1) + 1\}\forall x.x > 0 \rightarrow h(x) = h(x-1) + 1$$

Finally, the initial example of the paper, i.e. Formula ϕ_1, can be solved by the following quantified update application which KeY simplifies to true.

$$\{(\texttt{for}\ x_1;\ x_1 \geqslant 0;\ \mathit{next}(x_1) := x_1);(\texttt{for}\ x_2;\ x_2 \geqslant 0;\ \mathit{prev}(\mathit{next}(x_2)) := x_2)\}\phi_1$$

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

Java + JML

```
/*@ public normal_behavior
  @ requires next!=null && prev!=null && next!=prev
  @  && (\forall int k; true ; 0<=next[k] && next[k] < prev.length)
  @  && (\forall int l; 0<=l && l<next.length; next[l]==l);
  @ ensures (\forall int j; 0<=j && j<next.length; prev[next[j]]==j);
  @ assignable prev[*]; */
public void link(){
 /*@ loop_invariant (\forall int x; 0<=x && x <= i; prev[next[x]]==x)
                   && (0<=i && i<=next.length) ; modifies prev[*],i; @*/
 for(int i=0;i<next.length;i++){  prev[next[i]]=i; }
}
```

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 $x \leqslant i$

$\forall x : \texttt{int}.(x \leqslant -1 \vee x \geqslant 1 + i_0 \vee \texttt{get}_0(\texttt{prev(self)}, \texttt{acc}_{[]}(\texttt{next(self)}, x) \doteq x),$
$\forall x : \texttt{MyCls}.(\texttt{prevAtPre}(x) \doteq \texttt{prev}(x)),$
$\forall x : \texttt{MyCls}.(x \doteq \texttt{null} \vee \neg\texttt{created}(x) \vee \neg\texttt{a}(x) \doteq \texttt{null}),$
$\forall x : \texttt{MyCls}.(x \doteq \texttt{null} \vee \neg\texttt{created}(x) \vee \neg\texttt{next}(x) \doteq \texttt{null}),$
$\forall x : \texttt{MyCls}.(x \doteq \texttt{null} \vee \neg\texttt{created}(x) \vee \neg\texttt{prev}(x) \doteq \texttt{null}),$
$\forall x : \texttt{int}.\texttt{acc}_{[]}(\texttt{next(self)}, x) \geqslant 0),$
$\forall x : \texttt{int}.\texttt{acc}_{[]}(\texttt{next(self)}, x) \leqslant -1 + \texttt{length}(\texttt{prev(self)})),$
$\forall x : \texttt{int}.(l \leqslant -1 \vee l \geqslant \texttt{length}(\texttt{next(self)}) \vee \texttt{acc}_{[]}(\texttt{next(self)}, x) \doteq x),$
$\ldots \Longrightarrow \ldots$

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 '...'

$\ldots$
$\{\texttt{for } x : \texttt{MyCls};\ (\texttt{next}(x) \doteq \texttt{null} \wedge \neg\texttt{a}(x) \doteq \texttt{null} \wedge \ldots);\ \texttt{created}(x) := \texttt{false}\}$
$\{\texttt{for } x : \texttt{MyCls};\ (\texttt{a}(x) \doteq 0 \wedge \neg x \doteq \texttt{null});\ \texttt{created}(x) := \texttt{false}\}$
$\{\texttt{for } x : \texttt{int};\ (b \geqslant 1 + x \wedge x \leqslant -1);\ \texttt{acc}_{[]}(\texttt{next(self)}) := -1 + c_2\}$
$\{\texttt{for } x : \texttt{int};\ x \leqslant -1;\ i := \texttt{acc}_{[]}(\texttt{next(self)}) - c_0 * -1 + c_1\}$
$\{\texttt{for } x : \texttt{int};\ (x \geqslant 0 \wedge x \geqslant 1 + i_0);\ \texttt{acc}_{[]}(\texttt{next(self)}) := \texttt{length}(\texttt{prev(self)}) + c_0\}$
$\{\texttt{for } x : \texttt{int};\ (\texttt{acc}_{[]}(\texttt{next(self)}, x) = x \wedge x \leqslant i_0 \wedge \ldots);\ \texttt{get}_0(\texttt{prev(self)}, x) := x\}$

Fig. 3. A subset of generated updates satisfying the quantified formulas in Figure 2

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 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 a formal software verification and test generation project. Verification attempts often fail, i.e., they are interrupted by a timeout. For instance, 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. Our experiments show that our method can generate counter examples for formulas such as φ that SMT solvers cannot solve [18]. 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. This problem can be probably solved by an extension of updates which is future work.

References

1. Barrett, C.W., Tinelli, C.: CVC3. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 298–302. Springer, Heidelberg (2007)
2. Baumgartner, P., Fuchs, A., Tinelli, C.: Implementing the model evolution calculus. International Journal on Artificial Intelligence Tools 15(1), 21–52 (2006)
3. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software. LNCS (LNAI), vol. 4334. Springer, Heidelberg (2007)
4. Benhamou, F., Goualard, F.: Universally quantified interval constraints. In: Dechter, R. (ed.) CP 2000. LNCS, vol. 1894, pp. 67–82. Springer, Heidelberg (2000)
5. Blanchette, J.C.: Relational analysis of (co)inductive predicates (co)algebraic datatypes, and (co)recursive functions. In: Fraser, G., Gargantini, A. (eds.) TAP 2010. LNCS, vol. 6143, pp. 117–134. Springer, Heidelberg (2010)
6. Bradley, A.R., Manna, Z., Sipma, H.B.: What's decidable about arrays? In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 427–442. Springer, Heidelberg (2005)
7. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000)
8. de Moura, L., Bjørner, N.S.: Efficient E-matching for SMT solvers. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 183–198. Springer, Heidelberg (2007)
9. de Moura, L., Bjørner, N.S.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
10. Déharbe, D., Ranise, S.: Satisfiability solving for software verification. STTT 11(3), 255–260 (2009)
11. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: A theorem prover for program checking. Technical Report. J. ACM (2003)
12. Dutertre, B., de Moura, L.: The YICES SMT solver. Technical report, Computer Science Laboratory, SRI International (2006)
13. Ge, Y., Barrett, C.W., Tinelli, C.: Solving quantified verification conditions using satisfiability modulo theories. Ann. Math. Artif. Intell. 55(1-2), 101–122 (2009)
14. Ge, Y., de Moura, L.M.: Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 306–320. Springer, Heidelberg (2009)
15. Ghilardi, S.: Quantifier elimination and provers integration. Electr. Notes Theor. Comput. Sci. 86(1) (2003)

16. Giese, M.: Incremental closure of free variable tableaux. In: Goré, R.P., Leitsch, A., Nipkow, T. (eds.) IJCAR 2001. LNCS (LNAI), vol. 2083, pp. 545–560. Springer, Heidelberg (2001)
17. Gladisch, C.: Satisfiability solving and model generation for quantified first-order logic formulas. Karlsruhe Reports in Informatics, Fakultät für Informatik Institut für Theoretische Informatik, ITI (2010) ISSN: 2190-4782
18. Gladisch, C.: Test data generation for programs with quantified first-order logic specifications. In: Petrenko, A., Simão, A., Maldonado, J.C. (eds.) ICTSS 2010. LNCS, vol. 6435, pp. 158–173. Springer, Heidelberg (2010)
19. Harel, D., Kozen, D., Tiuryn, J.: Dynamic Logic. The MIT Press, London (2000)
20. KeY project homepage, `http://www.key-project.org/`
21. Kiniry, J.R., Morkan, A.E., Denby, B.: Soundness and completeness warnings in ESC/Java2. In: Proc. Fifth Int. Workshop Specification and Verification of Component-Based Systems, pp. 19–24 (2006)
22. Moskal, M.: Satisfiability Modulo Software. PhD thesis, University of Wrocław (2009)
23. Moskal, M., Lopuszanski, J., Kiniry, J.R.: E-matching for fun and profit. Electr. Notes Theor. Comput. Sci. 198(2), 19–35 (2008)
24. Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E., Rubio, A.: Challenges in satisfiability modulo theories. In: Baader, F. (ed.) RTA 2007. LNCS, vol. 4533, pp. 2–18. Springer, Heidelberg (2007)
25. Ricardo, C., Alexander, L., Nicolas, P.: *Automated Model Building.* Applied Logic Series, vol. 31. Springer, Heidelberg (2004)
26. Rümmer, P.: Sequential, parallel, and quantified updates of first-order structures. In: Hermann, M., Voronkov, A. (eds.) LPAR 2006. LNCS (LNAI), vol. 4246, pp. 422–436. Springer, Heidelberg (2006)
27. Rümmer, P., Shah, M.A.: Proving programs incorrect using a sequent calculus for java dynamic logic. In: Gurevich, Y., Meyer, B. (eds.) TAP 2007. LNCS, vol. 4454, pp. 41–60. Springer, Heidelberg (2007)

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 workshop 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 a 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 manipulate 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, and building bytecode static analyses. 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[1] (BCEL) and ASM[2] 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 run-time, 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. Those libraries are well adapted to instrument Java classes but lack important features essential for the design of static analyses. For instance, unlike Sawja,

[1] `http://jakarta.apache.org/bcel/`

[2] `http://asm.ow2.org/`

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. Sawja only targets static analysis tools and does not propose inverse transformations from IR to bytecode. 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 is foreseen for the future releases.

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.[3] JAVALIB handles all aspects of class files, including stackmaps (J2ME and JAVA 6) and JAVA 5 annotation attributes. It is made of three modules: $\boxed{\text{Javalib}}$, $\boxed{\text{JBasics}}$, and $\boxed{\text{JCode}}$[4].

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 make 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[5]. Fig. 1 describes the hierarchy formed by these

[3] JAVALIB is a sub-component of SAWJA, which, while 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.

[4] In the following, we use boxes around JAVALIB and SAWJA module names to make clickable links to the on-line API documentation.

[5] Polymorphic variants are a particular notion of enumeration which allows the sharing of constructors between types.

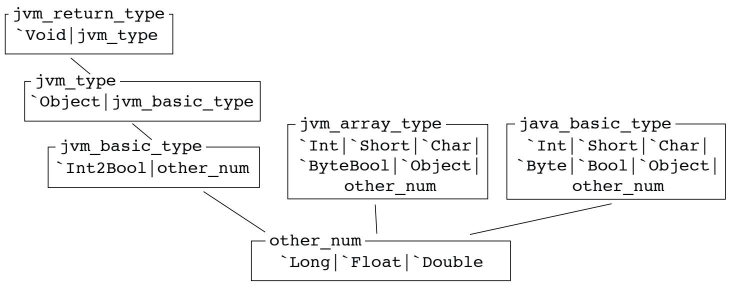


Fig. 1. Hierarchy of Java bytecode types. Links represent the subtyping relation enforced by polymorphic variants (for example, the type `jvm_type` is defined by `type jvm_type = [ |'Object |jvm_basic_type ]`).

types. This factorization principle applies in particular to the representation of op-codes: many 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[6]. 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

[6] The indexing time is compensated by a reduced stress on the garbage collector.

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 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

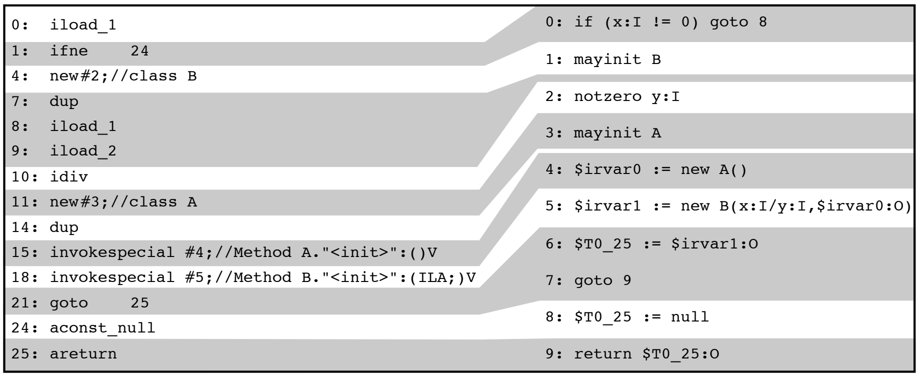
The JVM is a stack-based virtual machine and the intensive use of the operand stack makes it difficult to adapt standard static analysis techniques that have been first designed for more classic variable-based codes. Hence, several bytecode optimization and analysis tools work on a bytecode *intermediate 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).



```
0:  iload_1
1:  ifne    24
4:  new#2;//class B
7:  dup
8:  iload_1
9:  iload_2
10: idiv
11: new#3;//class A
14: dup
15: invokespecial #4;//Method A."<init>":()V
18: invokespecial #5;//Method B."<init>":(ILA;)V
21: goto    25
24: aconst_null
25: areturn
```

```
0: if (x:I != 0) goto 8
1: mayinit B
2: notzero y:I
3: mayinit A
4: $irvar0 := new A()
5: $irvar1 := new B(x:I/y:I,$irvar0:O)
6: $T0_25 := $irvar1:O
7: goto 9
8: $T0_25 := null
9: return $T0_25:O
```

Fig. 2. Example of bytecode (left) (obtained with `javap -c`) and its corresponding IR (right). Colors make explicit the boundaries of related code fragments.

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 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.[7] 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.

[7] 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]: the verification of the result of a static analysis, i.e., checking that it is indeed a fixpoint, 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, using dedicated temporary variables. 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 library[8], we compare our running time for transforming all classes with

[8] For scale reason, the Java runtime library measures are not shown here.

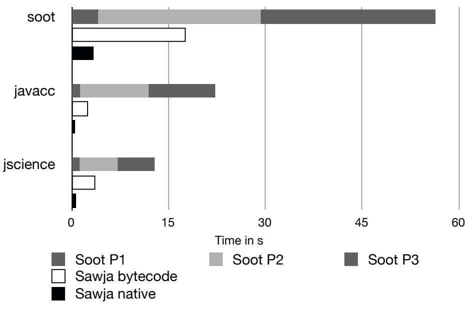


Fig. 3. SAWJA and SOOT IR generation times

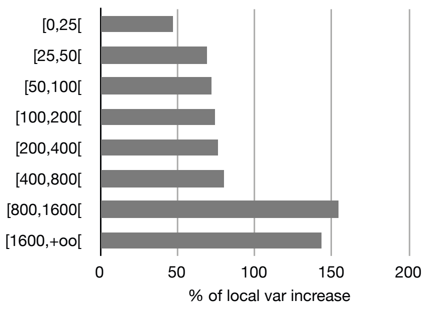


Fig. 4. SAWJA: local variable increase

the running time of SOOT. Here, we choose to generate with SOOT the Grimp representation of classes[9], 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 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 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,

[9] The SOOT transformation is without any optimisation option.

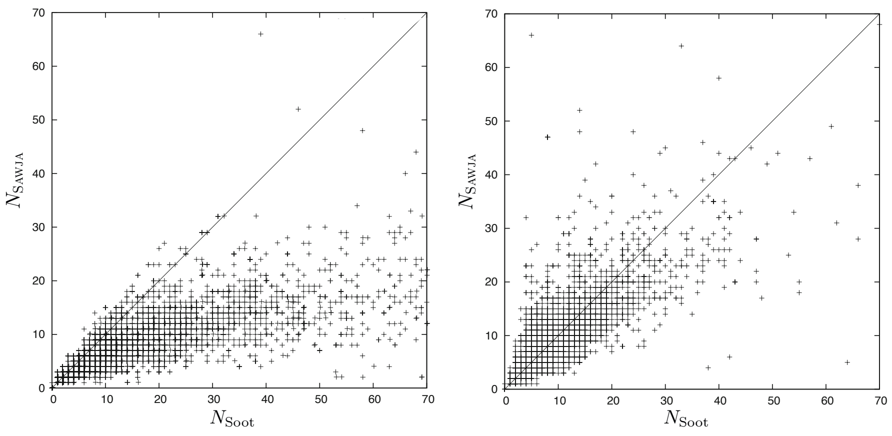


Fig. 5. Local variable increase ratio between Sawja and Soot

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 that reallocates local variables using use/def information (and hence increases the transformation time). Our transformation competes well, even when Soot uses this last optimization. We could probably improve this ratio using a similar packing, but this would require to iterate on the code.

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 objects 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 method resolution, 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.

Experimental Evaluation. We evaluate the precision and performances of the class analyses implemented in Sawja on several pieces of Java software[10] 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

[10] 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).

Table 1. Comparison of algorithms generating a program call graph (with Sawja and Wala): the algorithms of Sawja (CRA,RTA and XTA) are compared to Wala (W-RTA and W-0CFA) w.r.t 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.

		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

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 (r3767). Although precision is hard to compare[11], 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.

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]. It is a quite efficient tool, able to analyze a complete program of more than 3000 classes and 26000 methods to infer nullness annotations for fields, method signatures

[11] 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.

and local variables to prove the safety of 84% of dereferences in less than 2 minutes. 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 has been ported to the current version of SAWJA, improving its performances by 30% in our first tests; while being integrated in SAWJA, the class analyses presented in Section 4.2 rely on the underlying features and can be seen as use cases of SAWJA; and other small analyses (liveness, interval analyses, etc.) are also 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. Bacon, D.F., Sweeney, P.F.: Fast static analysis of C++ virtual function calls. In: Proc. of OOPSLA 1996, pp. 324–341 (1996)
2. Bicolano - web home, `http://mobius.inria.fr/bicolano`
3. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. In: Proc. of PLDI 2003, San Diego, California, USA, June 7–14, pp. 196–207. ACM Press, New York (2003)
4. Bravenboer, M., Smaragdakis, Y.: Strictly declarative specification of sophisticated points-to analyses. SIGPLAN Not. 44(10), 243–262 (2009)
5. Bryant, R.E.: Symbolic Boolean manipulation with ordered binary-decision diagrams. ACM Computing Survey 24(3), 293–318 (1992)
6. Burke, M.G., Choi, J., Fink, S., Grove, D., Hind, M., Sarkar, V., Serrano, M.J., Sreedhar, V.C., Srinivasan, H., Whaley, J.: The Jalapeño dynamic optimizing compiler for Java. In: Proc. of JAVA 1999, pp. 129–141. ACM, New York (1999)
7. Clerc, X.: Barista, `http://barista.x9c.fr/`
8. The Coq Proof Assistant, `http://coq.inria.fr/`
9. Dean, J., Grove, D., Chambers, C.: Optimization of object-oriented programs using static class hierarchy analysis. In: Olthoff, W. (ed.) ECOOP 1995. LNCS, vol. 952, pp. 77–101. Springer, Heidelberg (1995)
10. Demange, D., Jensen, T., Pichardie, D.: A provably correct stackless intermediate representation for Java bytecode. Research Report 7021, INRIA (2009), `http://www.irisa.fr/celtique/ext/bir/rr7021.pdf`

11. Ershov, A.P.: On programming of arithmetic operations. Commun. ACM 1(8), 3–6 (1958)
12. Grove, D., Chambers, C.: A framework for call graph construction algorithms. Toplas 23(6), 685–746 (2001)
13. Hubert, L.: A Non-Null annotation inferencer for Java bytecode. In: Proc. of PASTE 2008, pp. 36–42. ACM, New York (November 2008)
14. Hubert, L., Jensen, T., Monfort, V., Pichardie, D.: Enforcing secure object initialization in java. In: Gritzalis, D., Preneel, B., Theoharidou, M. (eds.) ESORICS 2010. LNCS, vol. 6345, pp. 101–115. Springer, Heidelberg (2010)
15. IBM: The T.J. Watson Libraries for Analysis (Wala), `http://wala.sourceforge.net`
16. Jensen, T., Pichardie, D.: Secure the clones: Static enforcement of policies for secure object copying. Technical report, INRIA (June 2010); Presented at OWASP (2010)
17. Leroy, X., Doligez, D., Garrigue, J., Rémy, D., Vouillon, J.: The Objective Caml system, Inria (May 2007), `http://caml.inria.fr/ocaml/`
18. Lhoták, O., Hendren, L.: Scaling java points-to analysis using SPARK. In: Wang, H. (ed.) CC 2003. LNCS, vol. 2622, pp. 153–169. Springer, Heidelberg (2003)
19. Lhoták, O., Hendren, L.: Evaluating the benefits of context-sensitive points-to analysis using a BDD-based implementation. ACM Trans. Softw. Eng. Methodol. 18(1) (2008)
20. Lindholm, T., Yellin, F.: The Java$^{\mathrm{TM}}$ Virtual Machine Specification, 2nd edn. Prentice Hall PTR, Englewood Cliffs (1999)
21. Livshits, V.B., Whaley, J., Lam, M.S.: Reflection analysis for java. In: Yi, K. (ed.) APLAS 2005. LNCS, vol. 3780, pp. 139–160. Springer, Heidelberg (2005)
22. Meyer, J., Downing, T.: Java Virtual Machine. O'Reilly Associates, Sebastopol (1997), `http://jasmin.sourceforge.net`
23. Morrison, D.R.: PATRICIA — Practical algorithm to retrieve information coded in alphanumeric. J. ACM 15(4) (1968)
24. Pagano, B., Andrieu, O., Moniot, T., Canou, B., Chailloux, E., Wang, P., Manoury, P., Colaço, J.L.: Experience report: using Objective Caml to develop safety-critical embedded tools in a certification framework. In: Proc. of ICFP, pp. 215–220. ACM, New York (2009)
25. Rose, E.: Lightweight bytecode verification. J. Autom. Reason. 31(3-4), 303–334 (2003)
26. Spoto, F.: Julia: A generic static analyser for the Java bytecode. In: Proc. of the Workshop FTfJP (2005)
27. Stata, R., Abadi, M.: A type system for Java bytecode subroutines. In: Proc. of POPL 1998, pp. 149–160. ACM Press, New York (1998)
28. Tip, F., Palsberg, J.: Scalable propagation-based call graph construction algorithms. In: Proc. of OOPSLA 2000, pp. 281–293. ACM Press, New York (October 2000)
29. Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., Sundaresan, V.: Soot - A Java bytecode optimization framework. In: Proc. of CASCON 1999 (1999)
30. Whaley, J.: Dynamic optimization through the use of automatic runtime specialization. Master's thesis, Massachusetts Institute of Technology (May 1999)

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:

– within atomic transactions, there are no calls to non–atomic methods;
– 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 when 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 [16]. The current version of the tool set, discussed in this paper, includes later extensions: (*i*) an inliner to abstract private methods [11], (*ii*) more general program models concerning exceptions, threads and open flow graphs [15,13], and (*iii*) a property translation from behavioural to structural properties [12,13]. The last extension allows local assumptions to be behavioural, whereas in the previous version they had to be structural. Further, we have unified the inputs and outputs to allow interoperability of the individual tools, and have started 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 [9]. 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. Another example of a successful system for the non–compositional verification of temporal safety properties, applied to C programs, is ESP [8]. This system combines a number of scalable program analyses to achieve precise tracking (simulation) of a given property on multiple stateful values (such as file handles), identified through user–defined source code patterns.

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 [18]. 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 [14].

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. In particular, 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 the fragment of the modal μ-calculus [17] with boxes and greatest fixed-points only is used. This temporal logic is capable of characterising simulation (*cf.* [14]) 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 [17]. 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.* [23]).

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.

```
class Number {
   public static boolean even(int n){
      if (n == 0)
         return true;
      else
         return odd(n-1);
   }

   public static boolean odd(int n){
      if (n == 0)
         return false;
      else
         return even(n-1);

   }}
```

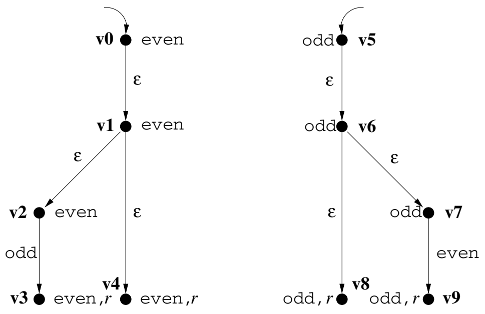


Fig. 1. A simple Java class and its flow graph

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.

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[1]. 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, $m_1 \; \mathsf{call} \; m_2$ invocation of method m_2 by method m_1, and $m_2 \; \mathsf{ret} \; m_1$ the corresponding return.

Definition 4. (Behaviour) *Let $\mathcal{G} = (\mathcal{M}, E) : I$ be a closed flow graph where $\mathcal{M} = (V, L, \rightarrow, A, \lambda)$. The* behaviour *of $\mathcal{G}$ is defined as the initialised model $b(\mathcal{G}) = (\mathcal{M}_b, E_b)$, where $\mathcal{M}_b = (S_b, L_b, \rightarrow_b, A_b, \lambda_b)$, 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 $\rightarrow_b \subseteq S_b \times L_b \times S_b$ is defined by the rules:*

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

[1] We only require I^- to contain methods that are not provided by I^+. This is different from our earlier work (*e.g.,* [14]), 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 [13]). Flow graph behaviour can alternatively be defined via *pushdown automata* (PDA) [14, 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 [20]; however, this requires the properties to be presented 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:

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

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 [19].

Example 4. For the flow graph from Example 1, the behavioural formula $\texttt{even} \Rightarrow \nu X.\ [\textsf{even call even}]\,\textsf{ff} \wedge [\tau]\,X$ 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 [15], 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 by Grumberg and Long [10] 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 of Definition 2, maximal models exist and are unique up to isomorphism (see [14]). 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 every model $\mathcal{M}'$, $\mathcal{M}' \leq \mathcal{M}$ iff $\mathcal{M}' \models \phi$. Due to this close connection between simulation and satisfaction, we obtain the following sound and complete verification principle [14]:

> *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 guarantees that no *false negatives* exist: if $\mathcal{M}_\phi \uplus \mathcal{M}_2 \models \psi$ fails, then there is 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. 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 [14] 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

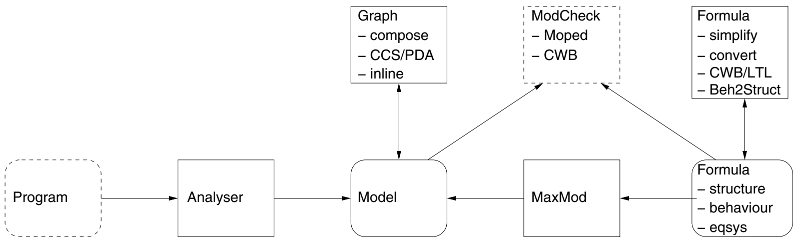


Fig. 2. The CVPP tool set architecture

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 [12]). 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:

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

In addition, we have also developed a "mixed" rule [14], where local structural assumptions are combined with global behavioural guarantees.

The presented proof rules are flexible, allowing 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 [21]).

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.
- *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 are therefore not included 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 [22].
- 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 [11]).
- 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 [20].

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:

```
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
```

The interface and structural and behaviour properties are written as follows in CVPP's input format:

```
    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))
```

5 Typical Verification Scenarios

Section 3 presented two 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 ($\mathsf{beh} - \mathsf{comp}$) 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. Typically, these contain information about private methods. In contrast, the abstract components and global properties are described in terms of the public interface. Therefore, the implementation details in the concrete components are abstracted away, by 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 [21].

5.3 Non–compositional Verification

The open system and modular verification scenarios above give rise to several non-modular verification tasks. In fact, CVPP can also be applied in a non-compositional setting. This is in particular useful when reasoning 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 case. 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. Given a Java application and a property specification (either behavioural or structural), this is done as follows:

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. 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 [21]. It automates procedure–modular verification of Java programs annotated with global and method–local specifications. ProMoVer was 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[2]. In the near future, we plan to develop wrapper tools for the other scenarios.

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 [21].

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

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

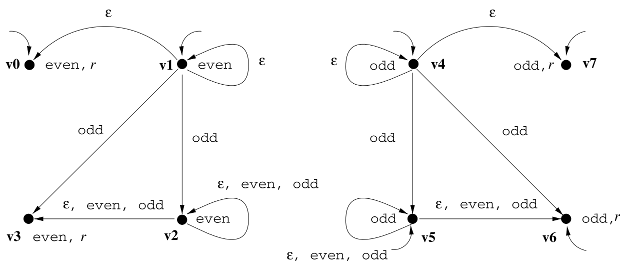


Fig. 3. Maximal flow graph for "the first call is not to method `even` itself"

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: $\mathtt{even} \Rightarrow \nu X. \, [\mathsf{even}] \, \mathsf{ff} \wedge [\epsilon] \, X$. 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 3. 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 [14,6]. 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 will provide support for other property specification formalisms, in particular security automata. 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. Alur, R., Arenas, M., Barcelo, P., Etessami, K., Immerman, N., Libkin, L.: First-order and temporal logics for nested words. In: Logic in Computer Science (LICS 2007), Washington, DC, USA, pp. 151–160. IEEE Computer Society, Los Alamitos (2007)
2. Alur, R., Chaudhuri, S.: Temporal reasoning for procedural programs. In: Barthe, G., Hermenegildo, M. (eds.) VMCAI 2010. LNCS, vol. 5944, pp. 45–60. Springer, Heidelberg (2010)
3. Alur, R., Etessami, K., Madhusudan, P.: A temporal logic of nested calls and returns. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 467–481. Springer, Heidelberg (2004)

4. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# Programming System: An Overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 49–69. Springer, Heidelberg (2005)
5. Boudol, G., Larsen, K.: Graphical versus logical specifications. Theoretical Computer Science 106, 3–20 (1992)
6. Chugunov, G., Fredlund, L.-Å., Gurov, D.: Model checking of multi-applet JavaCard applications. In: Smart Card Research and Advanced Application Conference (CARDIS 2002), pp. 87–95. USENIX Publications (2002)
7. Cleaveland, R., Parrow, J., Steffen, B.: A semantics based verification tool for finite state systems. In: International Symposium on Protocol Specification, Testing and Verification, pp. 287–302. North-Holland Publishing Co., Amsterdam (1990)
8. Das, M., Lerner, S., Seigle, M.: ESP: Path–sensitive program verification in polynomial time. In: Programming Language Design and Implementation (PLDI 2002), pp. 57–68. ACM, New York (2002)
9. Goldman, M., Katz, S.: MAVEN: Modular aspect verification. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 308–322. Springer, Heidelberg (2007)
10. Grumberg, O., Long, D.: Model checking and modular verification. ACM TOPLAS 16(3), 843–871 (1994)
11. Gurov, D., Huisman, M.: Interface abstraction for compositional verification. In: Software Engineering and Formal Methods (SEFM 2005), pp. 414–423 (2005)
12. Gurov, D., Huisman, M.: Reducing behavioural to structural properties of programs with procedures. In: Jones, N.D., Müller-Olm, M. (eds.) VMCAI 2009. LNCS, vol. 5403, pp. 136–150. Springer, Heidelberg (2009)
13. Gurov, D., Huisman, M.: Reducing behavioural to structural properties of programs with procedures (2010); Full version, available upon request
14. Gurov, D., Huisman, M., Sprenger, C.: Compositional verification of sequential programs with procedures. Information and Computation 206(7), 840–868 (2008)
15. Huisman, M., Aktug, I., Gurov, D.: Program models for compositional verification. In: Liu, S., Araki, K. (eds.) ICFEM 2008. LNCS, vol. 5256, pp. 147–166. Springer, Heidelberg (2008)
16. Huisman, M., Gurov, D., Sprenger, C., Chugunov, G.: Checking absence of illicit applet interactions: A case study. In: Wermelinger, M., Margaria-Steffen, T. (eds.) FASE 2004. LNCS, vol. 2984, pp. 84–98. Springer, Heidelberg (2004)
17. Kozen, D.: Results on the propositional μ-calculus. Theoretical Computer Science 27, 333–354 (1983)
18. Müller, P.: Modular Specification and Verification of Object-Oriented Programs. LNCS, vol. 2262. Springer, Heidelberg (2002)
19. Schneider, F.B.: Enforceable security policies. ACM Trans. Infinite Systems Security 3(1), 30–50 (2000)
20. Schwoon, S.: Model-Checking Pushdown Systems. PhD thesis, Technische Universität München (2002)
21. Soleimanifard, S., Gurov, D., Huisman, M.: Procedure–modular verification of control flow safety properties. In: Workshop on Formal Techniques for Java Programs (FTfJP 2010) (2010)
22. Vallée-Rai, R., Hendren, L., Sundaresan, V., Lam, P., Gagnon, E., Co, P.: Soot - a Java Optimization Framework. In: CASCON 1999, pp. 125–135 (1999)
23. Walukiewicz, I.: Completeness of Kozen's axiomatisation of the propositional mu-calculus. In: Logic in Computer Science (LICS 1995), pp. 14–24. IEEE, Los Alamitos (1995)

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.

Keywords: 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.[1] 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

$$A \rightarrow [\,p\,]\,B \tag{1}$$

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

[1] Use a field for each argument and a unique application method for functions; use fields to represent the content and getter/setter methods for references.

(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: $A \rightarrow \langle p \rangle B$.

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 (DL formula over IML program)
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 expressivity of higher-order logic. This allows to reason about programs independently of 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 $s_1 \oslash s_2$ and defined by

$$\begin{array}{rcll} \mathsf{dom}(s_1 \oslash s_2) & = & \mathsf{dom}(s_1) \cup \mathsf{dom}(s_2) & \\ (s_1 \oslash s_2)(\ell) & = & s_2(\ell) & \text{if } \ell \in \mathsf{dom}(s_2) \\ (s_1 \oslash s_2)(\ell) & = & s_1(\ell) & \text{otherwise} \end{array}$$

Intuitively, $s_1 \oslash s_2$ 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.[2] We write $\mathsf{C}(p)$ for the (closed) program obtained by substituting the (closed) program p to the hole () in the evaluation context C.

[2] 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 $\succ$.

$$\begin{array}{rcl}
(\lambda x.\, p)\ v \star s & \succ & [^v/_x]\ p \star s \\
\mathsf{fst}(v_1, v_2) \star s & \succ & v_1 \star s \\
\mathsf{snd}(v_1, v_2) \star s & \succ & v_2 \star s
\end{array}
\qquad
\begin{array}{rcll}
f(n_1, \ldots, n_k) \star s & \succ & \tilde{f}(n_1, \ldots, n_k) \star s & \\
\mathsf{if}\ n\ \mathsf{then}\ p_1\ \mathsf{else}\ p_2 \star s & \succ & p_1 \star s & (\text{if } n \neq 0) \\
\mathsf{if}\ 0\ \mathsf{then}\ p_1\ \mathsf{else}\ p_2 \star s & \succ & p_2 \star s &
\end{array}$$

$$v = v' \star s \ \succ \ \begin{cases} 1 \star s & \text{if } v \sim v' \\ 0 \star s & \text{if } v \not\sim v' \end{cases}
\qquad
\begin{array}{rcll}
\mathsf{ref}\ v \star s & \succ & \ell \star (s \olessthan \ell \leftarrow v) & (\text{if } \ell = \mathsf{alloc}(s)) \\
\ell := v \star s & \succ & 0 \star s \olessthan \ell \leftarrow v & (\text{if } \ell \in \mathsf{dom}(s)) \\
!\ell \star s & \succ & s(\ell) \star s & (\text{if } \ell \in \mathsf{dom}(s))
\end{array}$$

$$\begin{array}{rcll}
v_1\ v_2 \star s & \succ & 0 \star s & (\text{if } v_1 \text{ is not an abstration}) \\
f(v_1, \ldots, v_k) \star s & \succ & 0 \star s & (\text{if } v_i \notin \mathbb{Z} \text{ for some } i \in [1..k]) \\
\mathsf{if}\ v\ \mathsf{then}\ p_1\ \mathsf{else}\ p_2 \star s & \succ & 0 \star s & (\text{if } v \notin \mathbb{Z})
\end{array}$$

$$\begin{array}{rcll}
\mathsf{fst}(v) \star s & \succ & 0 \star s & (\text{if } v \text{ is not a pair}) \\
\mathsf{snd}(v) \star s & \succ & 0 \star s & (\text{if } v \text{ is not a pair})
\end{array}
\qquad
\begin{array}{rcll}
v := v' \star s & \succ & 0 \star s & (\text{if } v \notin \mathsf{dom}(s)) \\
!v \star s & \succ & 0 \star s & (\text{if } v \notin \mathsf{dom}(s))
\end{array}$$

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:

$$\begin{array}{rl} \mathsf{v} ::= & x \mid \alpha \mid n \mid \mathsf{f}(\mathsf{v}_1,\ldots,\mathsf{v}_n) \mid \mathsf{v}_1 = \mathsf{v}_2 \mid \lambda x.\, p \\ & \mid (\mathsf{v}_1,\ \mathsf{v}_2) \mid \mathsf{fst}(\mathsf{v}) \mid \mathsf{snd}(\mathsf{v}) \mid \mathsf{if}\ \mathsf{v}\ \mathsf{then}\ \mathsf{v}_1\ \mathsf{else}\ \mathsf{v}_2 \end{array}$$

(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 $\mathsf{let}\ x = \mathsf{v}\ \mathsf{in}\ p$.

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 the following rules:

$$\begin{array}{rcl@{\qquad}rcll}
\mathsf{f}(n_1, \dots, n_k) & \cong & \tilde{\mathsf{f}}(n_1, \dots, n_k) & \mathsf{if}\ n\ \mathsf{then}\ p\ \mathsf{else}\ p' & \cong & p & (n \neq 0) \\
 & & & \mathsf{if}\ 0\ \mathsf{then}\ p\ \mathsf{else}\ p' & \cong & p' & \\
(\lambda x.\, p)\ \mathsf{v} & \cong & [{}^{\mathsf{v}}/_{x}]\ p & & & & \\
 & & & \mathsf{v} = \mathsf{v} & \cong & 1 & \\
\mathsf{fst}((\mathsf{v}_1,\ \mathsf{v}_2)) & \cong & \mathsf{v}_1 & n = m & \cong & 0 & (n \neq m) \\
\mathsf{snd}((\mathsf{v}_1,\ \mathsf{v}_2)) & \cong & \mathsf{v}_2 & \alpha = \beta & \cong & 0 & (\alpha \neq \beta)
\end{array}$$

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.[3] 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:

$$\begin{array}{rcl}
A & ::= & X(\mathsf{v}_1, \dots, \mathsf{v}_k) \quad | \quad A \to B \quad | \quad \forall x.\ A \quad | \quad \forall X^k.\ A \\
 & | & I(\mathsf{v}) \quad | \quad \nu\alpha.\ A \quad | \quad [p\ \ \mathsf{as}\ x]\,A \quad | \quad \{u\}A
\end{array}$$

[3] Integer addition as well as all standard arithmetic operations are included in the set of primitive functions (see Sect. 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 $FV(A)$ ($FN(A)$).

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. 2).

$$\begin{array}{rcll}
I(0) & \cong & \bot\ (\equiv \forall X.X) & \\
I(n) & \cong & \top\ (\equiv \forall X.X \to X) & n \neq 0
\end{array}$$

Decomposition of boxes

$$\begin{array}{rcll}
[\mathsf{C}_{\mathsf{se}}(p)\ \ \mathsf{as}\ x]\,A & \cong & [p\ \ \mathsf{as}\ y]\,[\mathsf{C}_{\mathsf{se}}(y)\ \ \mathsf{as}\ x]\,A & y \notin FV(\mathsf{C}_{\mathsf{se}}(p), A, x) \\
[\mathsf{ref}\ \mathsf{v}\ \ \mathsf{as}\ x]\,A & \cong & \nu\alpha.\{\alpha\ := \mathsf{v}\}[^{\alpha}/_{x}]\ A & \alpha \notin FN(A, \mathsf{v}) \\
[\mathsf{v}_1\ := \mathsf{v}_2]A & \cong & \{\mathsf{v}_1\ := \mathsf{v}_2\}A & \\
[\mathsf{v}\ \ \mathsf{as}\ x]\,A & \cong & [^{\mathsf{v}}/_{x}]\ A &
\end{array}$$

Propagation of updates

$$\begin{array}{rcll}
\{u\}I(\mathsf{v}) & \cong & I(\mathsf{v}) & \\
\{u\}(A \to B) & \cong & \{u\}A \to \{u\}B & \\
\{u\}\forall x.A & \cong & \forall x.\{u\}A & x \notin FV(u) \\
\{u\}\forall X.A & \cong & \forall X.\{u\}A & \\
\{u\}\nu\alpha.A & \cong & \nu\alpha.\{u\}A & \alpha \notin FN(u) \\
\{u\}\{u'\}A & \cong & \{u\ ;\ u'\}A & \\
\{u\}[!\mathsf{v}\ \ \mathsf{as}\ x]\,A & \cong & [\{u\}!\mathsf{v}\ \ \mathsf{as}\ x]\,\{u\}A & x \notin FV(u)
\end{array}$$

Propagation of νs

$$\begin{array}{rcll}
\forall X^n.\ \nu\alpha.A & \cong & \nu\alpha.\forall X^n.\ A & \\
[p\ \ \mathsf{as}\ x]\,\nu\alpha.A & \cong & \nu\alpha.[p\ \ \mathsf{as}\ x]\,A & \alpha \notin FN(p) \\
\nu\alpha.\nu\beta.A & \cong & \nu\beta.\nu\alpha.A &
\end{array}$$

Fig. 2. 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 $\forall x.\nu\alpha.A \not\rightarrow \nu\alpha.\forall x.A$ 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 $\nu\alpha.A \not\rightarrow A$ even if $\alpha \notin FN(A)$.

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. 3. Specific deduction rules

The specific rules of our system (see Fig. 3) include:

– Left and right rules for symbolic evaluation, expressing that computationally equivalent formulas (via symbolic evaluation) are logically equivalent.[4]
– 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.

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. The store $\sigma(s)$ is defined by $\mathsf{dom}(\sigma(s)) = \sigma(\mathsf{dom}(s))$ and $\sigma(s)(\ell) = \sigma(s(\sigma^{-1}(\ell)))$ for $\ell \in \mathsf{dom}(\sigma(s))$.

[4] For general programs the application of these rules leads to undecidable premisses. In a concrete implementation one would use heuristics that, for example, limit the number of evaluation steps.

Propositional functions. Let $f : \mathcal{V}^k \to \mathfrak{P}(\$)$ 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 $v_1, \ldots v_k \in \mathcal{V}$, $s \in \$$ 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 $f : \mathcal{V}^k \to \mathfrak{P}(\$)$ 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:

$$[\![\emptyset]\!]_\rho = \emptyset \qquad \text{and} \qquad [\![u\,;\mathsf{v}_1 := \mathsf{v}_2]\!]_\rho = [\![u]\!]_\rho \oslash [\![\mathsf{v}_1]\!]_\rho \leftarrow [\![\mathsf{v}_2]\!]_\rho$$

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. 4.

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.

$s \models X(\mathsf{v}_1, \dots, \mathsf{v}_k)[\rho]$	iff	$s \in \rho(X)([\![\mathsf{v}_1]\!]_\rho, \dots, [\![\mathsf{v}_k]\!]_\rho)$
$s \models I(\mathsf{v})[\rho]$	iff	$[\![\mathsf{v}]\!]_\rho \neq 0$
$s \models (A \to B)[\rho]$	iff	$s \models A[\rho]$ implies $s \models B[\rho]$
$s \models (\forall x. A)[\rho]$	iff	for all $v \in \mathcal{V}_s$, $s \models A[\rho\,;\, x \mapsto v]$
$s \models (\forall X^k.\ A)[\rho]$	iff	for all $f \in \mathcal{F}^k_{\mathcal{P}}$, $s \models A[\rho\,;\, X \mapsto f]$
$s \models (\nu\alpha. A)[\rho]$	iff	$(s \otimes \mathsf{alloc}(s) \leftarrow 0) \models A[\rho\,;\, \alpha \mapsto \mathsf{alloc}(s)]$
$s \models (\{u\}A)[\rho]$	iff	$s \otimes [\![u]\!]_\rho \models A[\rho]$
$s \models ([p\ \ \mathsf{as}\ x]\ A)[\rho]$	iff	for all $s' \in \$$ and $v \in \mathcal{V}_{s'}$, $p[\rho] \star s \succ^* v \star s'$ implies $s' \models A[\rho\,;\, x \mapsto v]$

Fig. 4. Satisfiability of formulas

Theorem 1 (Correctness of the system). *If the sequent $\Gamma \vdash \Delta$ is derivable in the system, then for all well-formed stores $s \in \$$ and for all valuations ρ that are well-formed in s, we have $s \models (\Gamma \vdash \Delta)[\rho]$.*

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

$$\begin{aligned} cc \quad \equiv \quad & \lambda n.\ \mathsf{let}\ c = (\mathsf{let}\ r = \mathsf{ref}\ 0\ \mathsf{in}\ \lambda x.\ \ r := !r + 1\,;\, !r)\ \mathsf{in} \\ & \qquad \mathsf{let}\ aux = \mathsf{fix}\ (\lambda f n.\ \mathsf{if}\ n = 0\ \mathsf{then}\ c\,0\ \mathsf{else}\ (c\,0\,;\, f\ (n-1)))\ \mathsf{in} \\ & \qquad aux\ n \end{aligned}$$

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. 5. 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 part of the proof partially evaluates the program $cc\ n$ to make the inner reference visible at the top level.

$$
\begin{prooftree}
\AxiomC{}
\RightLabel{Ax}
\UnaryInfC{$\vdash\ I(1)$}
\RightLabel{$\cong$}
\UnaryInfC{$\vdash\ \{\alpha\ :=\ k+1\}I(1)$}
\RightLabel{$\cong\ \times 2$}
\UnaryInfC{$\vdash\ \{\alpha\ :=\ k+1\}[k+1=k+1\ \ \mathsf{as}\ b]\ I(b)$}
\RightLabel{$\cong$}
\UnaryInfC{$\vdash\ \{\alpha\ :=\ k+1\}[!\alpha=k+1\ \ \mathsf{as}\ b]\ I(b)$}
\RightLabel{$\cong$}
\UnaryInfC{$\vdash\ \{\alpha\ :=\ k\}[(\alpha:=k+1;!\alpha)=k+1\ \ \mathsf{as}\ b]\ I(b)$}
\RightLabel{$\cong$}
\UnaryInfC{$\vdash\ \{\alpha\ :=\ k\}[(\alpha:=!\alpha+1;!\alpha)=k+1\ \ \mathsf{as}\ b]\ I(b)$}
\RightLabel{$\cong$}
\UnaryInfC{$\vdash\ \{\alpha\ :=\ k\}[\underline{\mathsf{aux}'}\ 0=k+1\ \ \mathsf{as}\ b]\ I(b)$}
\RightLabel{$\forall$R}
\UnaryInfC{$\vdash\ \forall k.\ \{\alpha\ :=\ k\}[\underline{\mathsf{aux}'}\ 0=k+1\ \ \mathsf{as}\ b]\ I(b)$}
\AxiomC{$[\Pi_3]$}
\noLine
\UnaryInfC{$\vdots$}
\UnaryInfC{$RH\ \vdash\ RS$}
\RightLabel{Ind}
\BinaryInfC{$\vdash\ \forall n:\mathsf{Nat}.\ \forall k.\ \{\alpha\ :=\ k\}[\underline{\mathsf{aux}'}\ n=n+k+1\ \ \mathsf{as}\ b]\ I(b)$}
\AxiomC{$[\Pi_2]$}
\noLine
\UnaryInfC{$\vdots$}
\UnaryInfC{$L,\ \mathsf{Nat}(n)\ \vdash\ C$}
\RightLabel{Cut}
\BinaryInfC{$\mathsf{Nat}(n)\ \vdash\ \{\alpha\ :=\ 0\}[\underline{\mathsf{aux}'}\ n=n+1\ \ \mathsf{as}\ b]\ I(b)$}
\RightLabel{νR}
\UnaryInfC{$\mathsf{Nat}(n)\ \vdash\ \nu\alpha.\{\alpha\ :=\ 0\}[\underline{\mathsf{aux}'}\ n=n+1\ \ \mathsf{as}\ b]\ I(b)$}
\RightLabel{$\cong$}
\UnaryInfC{$\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)$}
\RightLabel{$\cong$}
\UnaryInfC{$\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)$}
\RightLabel{Let $+\cong$}
\UnaryInfC{$\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)$}
\RightLabel{Let $\times\ 2+\cong$}
\UnaryInfC{$\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)$}
\RightLabel{$\cong$}
\UnaryInfC{$\mathsf{Nat}(n)\ \vdash\ [\underline{\mathsf{cc}}\ n=n+1\ \ \mathsf{as}\ b]\ I(b)$}
\RightLabel{$\forall$R}
\UnaryInfC{$\vdash\ \forall n:\mathsf{Nat}.\ [\underline{\mathsf{cc}}\ n=n+1\ \ \mathsf{as}\ b]\ I(b)$}
\end{prooftree}
$$

$$
\begin{array}{llll}
\text{where} & RS & \equiv & \forall k.\{\alpha\ :=\ k\}[\underline{\mathsf{aux}'}(n+1)=n+k+2\ \ \mathsf{as}\ b]\ I(b) \\
 & RH & \equiv & \mathsf{Nat}(n),\ \forall k.\{\alpha\ :=\ k\}[\underline{\mathsf{aux}'}\ n=n+k+1\ \ \mathsf{as}\ b]\ I(b) \\
 & L & \equiv & \forall n:\mathsf{Nat}.\ \forall k.\ \{\alpha\ :=\ k\}[\underline{\mathsf{aux}'}\ n=n+k+1\ \ \mathsf{as}\ b]\ I(b) \\
 & C & \equiv & \{\alpha\ :=\ 0\}[\underline{\mathsf{aux}'}\ n=n+1\ \ \mathsf{as}\ b]\ I(b)
\end{array}
$$

Fig. 5. Π_1: Derivation of the specification

The proof of the recursive step (Π_3) is done using only $\cong, \forall$R, $\forall$L. The sequent $L,\ \mathsf{Nat}(n)\ \vdash\ C$ is trivially proved (Π_2) with instances of the $\forall$L rule.

$$
\begin{array}{lll}
\underline{\mathsf{c}} & \equiv & \mathsf{let}\ r=\mathsf{ref}\ 0\ \mathsf{in}\ \lambda x.\ r:=!r+1\ ;!r \\
\underline{\mathsf{aux}} & \equiv & \mathsf{fix}\ (\lambda fn.\ \mathsf{if}\ n=0\ \mathsf{then}\ c\ 0\ \mathsf{else}\ (c\ 0\ ;f\ (n-1))) \\
\underline{\mathsf{cc}} & \equiv & \lambda n.\ \mathsf{let}\ c=\underline{\mathsf{c}}\ \mathsf{in}\ \mathsf{let}\ aux=\underline{\mathsf{aux}}\ \mathsf{in}\ aux\ n \\
\underline{\mathsf{aux}'} & \equiv & \mathsf{fix}\ (\lambda fn.\ \mathsf{if}\ n=0\ \mathsf{then}\ (\lambda x.\ \alpha:=!\alpha+1\ ;!\alpha)\ 0 \\
 & & \qquad\qquad\qquad\qquad \mathsf{else}\ \ (\lambda x.\ \alpha:=!\alpha+1\ ;!\alpha)\ 0\ ;f\ (n-1))
\end{array}
$$

Fig. 6. Shortcuts

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. Balser, M., Reif, W., Schellhorn, G., Stenzel, K., Thums, A.: Formal system development with KIV. In: Maibaum, T. (ed.) FASE 2000. LNCS, vol. 1783, pp. 363–366. Springer, Heidelberg (2000)
2. Baro, S.: Introduction to PAF!, a proof assistant for ML programs verification. In: Berardi, S., Coppo, M., Damiani, F. (eds.) TYPES 2003. LNCS, vol. 3085, pp. 51–65. Springer, Heidelberg (2004)
3. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software. LNCS (LNAI), vol. 4334. Springer, Heidelberg (2007)
4. Chlipala, A., Malecha, G., Morrisett, G., Shinnar, A., Wisnesky, R.: Effective interactive proofs for higher-order imperative programs. In: ICFP 2009: Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (September 2009)
5. Filliâtre, J.-C.: Why: a multi-language multi-prover verification tool. Research Report 1366, LRI, Université Paris Sud (March 2003)
6. Harel, D., Kozen, D., Tiuryn, J.: Dynamic Logic. In: Foundations of Computing. MIT Press, Cambridge (October 2000)
7. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM 12(10), 576–580 (1969)
8. Kanig, J., Filliâtre, J.-C.: Who: A Verifier for Effectful Higher-order Programs. In: ACM SIGPLAN Workshop on ML, Edinburgh, Scotland (August 2009)
9. Krivine, J.L.: Lambda-calculus, types and models. Masson (1993)
10. Régis-Gianas, Y., Pottier, F.: A hoare logic for call-by-value functional programs. In: Audebaud, P., Paulin-Mohring, C. (eds.) MPC 2008. LNCS, vol. 5133, pp. 305–335. Springer, Heidelberg (2008)
11. Rümmer, P.: Sequential, parallel, and quantified updates of first-order structures. In: Hermann, M., Voronkov, A. (eds.) LPAR 2006. LNCS (LNAI), vol. 4246, pp. 422–436. Springer, Heidelberg (2006)

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`, we do not want to make use of the fact that there is only one implementation of

Java

```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);
  }
}
```

```java
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

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*:

$$\mathit{pre}\colon \texttt{this.}\mathit{inv} \wedge 0 \leq \texttt{i} \wedge \texttt{i} < \texttt{this.size()} \qquad \mathit{post}\colon \texttt{res} \not\doteq \texttt{null}$$

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:

$$\begin{aligned} &\mathit{exactInstance}_{\texttt{ArrayList}}(\texttt{this}) \\ &\rightarrow \big(\texttt{this.}\mathit{inv} \;\leftrightarrow\; \texttt{this.a} \not\doteq \texttt{null} \wedge \texttt{this.n} < \texttt{this.a.length} \\ &\qquad\qquad\qquad\quad \wedge\; \forall \mathit{Int}\, i; (0 \leq i \wedge i < \texttt{this.n} \rightarrow \texttt{this.a[}i\texttt{]} \not\doteq \texttt{null})\big) \end{aligned} \tag{1}$$

For a type A and an expression e, the formula $\mathit{exactInstance}_A(\texttt{e})$ 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:

$$\mathit{pre}\colon \texttt{l} \not\doteq \texttt{null} \wedge \texttt{l.}\mathit{inv} \wedge 0 < \texttt{l.size()} \qquad \mathit{post}\colon \texttt{res} \not\doteq \texttt{null}$$

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:

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

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:

$$\mathit{pre}\colon \texttt{l} \not\doteq \texttt{null} \land \texttt{l.}\mathit{inv} \land 0 < \texttt{l.size()} \land (\texttt{this}, \texttt{x}) \mathrel{\dot\notin} \texttt{l.}\mathit{locs}$$

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 $\texttt{l.}\mathit{inv} \land 0 < \texttt{l.size()}$ 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:

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

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 `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 $(\mathit{Object}, \mathit{Field})$ 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 `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 selectOfStore depicted in Fig. 2: The value $\mathit{select}_A(\mathit{store}(h, o, f, t), o', f')$ of a location (o, f) retrieved from a modified heap $\mathit{store}(h, o, f, t)$ depends on whether

$$select_A(store(h,o,f,t),o',f') \leadsto \quad \text{(selectOfStore)}$$
$$if(o \doteq o' \wedge f \doteq f')then(cast_A(t))else(select_A(h,o',f'))$$
$$select_A(anon(h,s,h'),o,f) \leadsto \quad \text{(selectOfAnon)}$$
$$if(((o,f)\,\dot{\in}\,s \wedge f \not\doteq created) \vee (o,f)\,\dot{\in}\,freshLocs(h))$$
$$then(select_A(h',o,f))$$
$$else(select_A(h,o,f))$$
$$cast_A(t) \leadsto t \qquad \text{for } t \in Trm_\Sigma^{A'} \text{ and } A' \sqsubseteq A \quad \text{(cast)}$$
$$(o,f)\,\dot{\in}\,freshLocs(h) \leadsto \quad \text{(inFreshLocs)}$$
$$o \not\doteq \mathtt{null} \wedge select_{Boolean}(h,o,created) \doteq FALSE$$
$$[\mathtt{a}\ \mathtt{=}\ t\mathtt{;}\ldots]\varphi \leadsto \{\mathtt{a} := t\}[\ldots]\varphi \quad \text{(assignLocal)}$$
$$[o\mathtt{.}f\ \mathtt{=}\ t\mathtt{;}\ldots]\varphi \leadsto \{\mathtt{heap} := store(\mathtt{heap},o,f,t)\}[\ldots]\varphi \quad \text{(assignField)}$$

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

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 $o\mathtt{.}created$ 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 $\dot{\emptyset}$, $\dot{\cup}$, $\dot{\cap}$, $\dot{\setminus}$, $\dot{\in}$, $\dot{\subseteq}$, *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

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

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 $\mathit{wellFormed}(\mathtt{heap})$ 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 $[\mathtt{p;q}]\varphi$, we handle the statement `p` first, and leave the formula $[\mathtt{q}]\varphi$ 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 $\mathtt{a} := t$. 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, $\mathtt{o} \in \mathcal{PV}$ is a local variable of a reference type, $\mathtt{f} : \mathit{Field} \in \mathcal{F}$ is a constant symbol, and $\mathtt{a} : \mathit{Int} \in \mathcal{PV}$ 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 $\mathtt{heap} := \mathit{store}(\mathtt{heap}, \mathtt{o}, \mathtt{f})$ 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 $\mathtt{a} \doteq 0$ as a substitution, resulting in an obviously valid formula. Hence, we have proven that the original formula is valid as well.

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

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* $\mathtt{hPre} : \mathit{Heap} \in \mathcal{PV}$*; and where* $pre, post \in \mathit{Fma}_\Sigma$*,* $mod \in \mathit{Trm}_\Sigma^{\mathit{LocSet}}$*, and* $\tau \in \{\mathit{partial}, \mathit{total}\}$*.*

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* $\mathit{CorrectMethodContract}(mct, B) \in \mathit{Fma}_\Sigma$ *is defined as*

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

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

- *reachableState is the formula*

$$\begin{aligned}
&\mathit{wellFormed}(\mathtt{heap}) \land \mathtt{this} \not\doteq \mathtt{null} \land \mathtt{this}.\mathit{created} \doteq \mathit{TRUE} \\
&\land \bigwedge_{i \in \{1,\ldots,n\},\ \mathtt{p}_i : A \text{ for some } A \sqsubseteq \mathit{Object}} (\mathtt{p}_i \doteq \mathtt{null} \lor \mathtt{p}_i.\mathit{created} \doteq \mathit{TRUE})
\end{aligned}$$

- *frame is the formula*

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

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, $\tau = \mathit{total}$, and $B = \texttt{ArrayList}$, we get the following instance of *CorrectMethodContract*:

$$\begin{aligned}&\texttt{this.}\mathit{inv} \land 0 \leq \texttt{i} \land \texttt{i} < \texttt{this.size()} \land \mathit{wellFormed}(\texttt{heap})\\ &\land \texttt{this} \not\doteq \texttt{null} \land \texttt{this.}\mathit{created} \doteq \mathit{TRUE} \land \mathit{exactInstance}_{\texttt{ArrayList}}(\texttt{this})\\ &\to \{\texttt{hPre} := \texttt{heap}\}\langle\texttt{res = this.get(i);}\rangle(\texttt{res} \not\doteq \texttt{null} \land \mathit{frame})\end{aligned}$$

where *frame* with a modifies clause $\mathit{mod} = \dot{\emptyset}$ 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)

$$\frac{\begin{array}{l}\Gamma \Rightarrow \{u\}\{w\}(\mathit{pre} \land \mathit{reachableState}),\ \Delta\\ \Gamma \Rightarrow \{u\}\{w\}\{\texttt{hPre} := \texttt{heap}\}\{v\}(\mathit{post} \land \mathit{reachableState}' \to [\![\ldots]\!]\varphi),\ \Delta\end{array}}{\Gamma \Rightarrow \{u\}[\![\texttt{r = o.m(}\texttt{p}'_1, \ldots, \texttt{p}'_n\texttt{);}\ \ldots]\!]\varphi,\ \Delta}$$

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* $\tau = \mathit{total}$ *if the modality* $[\![\cdot]\!]$ *is* $\langle\cdot\rangle$*, and where* τ *does not matter otherwise; and where* `this`, $\texttt{p}_1, \ldots, \texttt{p}_n$, `res` *and* `hPre` *do not occur in the formula* $[\![\texttt{r = o.m(}\texttt{p}'_1, \ldots, \texttt{p}'_n\texttt{);}\ \ldots]\!]\varphi$

– $\texttt{p}'_1, \ldots, \texttt{p}'_n$ *are terms*

– $\mathit{reachableState} \in \mathit{Fma}_\Sigma$ *is as in Def. 3, and* $\mathit{reachableState}'$ *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* $\mathit{wellFormed}(\texttt{heap})$ *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:*

$$\begin{aligned}
&\mathit{pre} \land \mathit{reachableState} \land \mathit{exactInstance}_B(\mathtt{this}) \\
&\to \mathtt{this}.\mathit{obs}(\mathtt{p}_1, \ldots, \mathtt{p}_n) \\
&\qquad \equiv \{\mathtt{heap} := \mathit{anon}(\mathtt{heap}, \mathit{allLocs} \mathbin{\dot{\setminus}} \mathit{dep}, h)\}\big(\mathtt{this}.\mathit{obs}(\mathtt{p}_1, \ldots, \mathtt{p}_n)\big)
\end{aligned}$$

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 $B = \mathtt{ArrayList}$, we get the following instance of *CorrectDependencyContract*:

$$\begin{aligned}
&\mathtt{this}.\mathit{inv} \land \mathit{wellFormed}(\mathtt{heap}) \land \mathtt{this} \not\doteq \mathtt{null} \land \mathtt{this}.\mathit{created} \doteq \mathit{TRUE} \\
&\land \mathit{exactInstance}_{\mathtt{ArrayList}}(\mathtt{this}) \\
&\to \big(\mathtt{this}.\mathit{inv} \leftrightarrow \{\mathtt{heap} := \mathit{anon}(\mathtt{heap}, \mathit{allLocs} \mathbin{\dot{\setminus}} \mathtt{this}.\mathit{locs}, h)\}(\mathtt{this}.\mathit{inv})\big)
\end{aligned}$$

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* $f_1, \ldots, f_m \in \{\mathit{store}, \mathit{anon}\}$, $h^{base} \in \mathit{Trm}_\Sigma^{Heap}$
- $\texttt{o} \in \mathit{Trm}_\Sigma^A$ *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,* $\mathit{mod} = \mathit{allLocs} \mathbin{\dot{\setminus}} \mathit{dep}$
- $\mathit{reachableState}, \mathit{frame} \in \mathit{Fma}_\Sigma$ *are as in Def. 3,* $w \in \mathit{Upd}_\Sigma$ *is as in Def. 4*
- $\mathit{noDeallocs} \in \mathit{Fma}_\Sigma$ *is the formula*

$$\begin{aligned}&\mathit{freshLocs}(\texttt{heap}) \mathrel{\dot{\subseteq}} \mathit{freshLocs}(\texttt{hPre})\\ &\wedge\ \texttt{null}.\mathit{created} \doteq \{\texttt{heap} := \texttt{hPre}\}\texttt{null}.\mathit{created}\end{aligned}$$

- *guard is the formula*

$$\begin{aligned}\{w\}\big(&\{\texttt{heap} := h^{base}\}(\mathit{pre} \wedge \mathit{reachableState})\\ &\wedge \{\texttt{hPre} := h^{base} \parallel \texttt{heap} := h^{new}\}(\mathit{frame} \wedge \mathit{noDeallocs})\big)\end{aligned}$$

- *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 $obs(h^{base}, \mathtt{o}, \mathtt{p}'_1, \ldots, \mathtt{p}'_n)$ 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 $inv(\mathtt{heap}, \mathtt{l})$ is true. To establish the precondition of the method call `l.get(0)` in the body of `m`, we need to establish that $inv(store(\mathtt{heap}, \mathtt{this}, \mathtt{x}, t), \mathtt{l})$ also holds (for some term t). Modularity deters us from using ([1]) to deduce this. Instead, we apply useDependencyContract, with $obs = inv$ and $h^{base} = \mathtt{heap}$. We get the following instantiation for *guard* (already slightly simplified):

$$\begin{aligned} &inv(\mathtt{heap}, \mathtt{l}) \wedge wellFormed(\mathtt{heap}) \wedge \mathtt{l} \not\doteq \mathtt{null} \wedge \mathtt{l}.created \doteq TRUE \\ &\wedge \forall Object\ o; \forall Field\ f; \big((o, f) \mathrel{\dot\in} \big((allLocs \mathbin{\dot\setminus} locs(\mathtt{heap}, \mathtt{l})) \mathbin{\dot\cup} freshLocs(\mathtt{heap})\big) \\ &\qquad\qquad\qquad\qquad \vee\ select_{Any}(store(\mathtt{heap}, \mathtt{this}, \mathtt{x}, t), o, f) \\ &\qquad\qquad\qquad\qquad\quad \doteq select_{Any}(\mathtt{heap}, o, f)\big) \ \wedge noDeallocs \end{aligned}$$

As the only location changed between the two heaps is $(\mathtt{this}, \mathtt{x})$, and as the precondition of `m` guarantees that $(\mathtt{this}, \mathtt{x}) \mathrel{\dot\notin} locs(\mathtt{heap}, \mathtt{l})$ holds, we can prove that the instantiation of *guard* is satisfied. This allows us to use the instantiation of *equal*, namely $inv(store(\mathtt{heap}, \mathtt{this}, \mathtt{x}, t), \mathtt{l}) \leftrightarrow inv(\mathtt{heap}, \mathtt{l})$, to prove that $inv(store(\mathtt{heap}, \mathtt{this}, \mathtt{x}, t), \mathtt{l})$ 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.

References

1. Barnett, M., DeLine, R., Fähndrich, M., Leino, K.R.M., Schulte, W.: Verification of object-oriented programs with invariants. Journal of Object Technology (JOT) 3(6), 27–56 (2004)
2. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software. LNCS (LNAI), vol. 4334. Springer, Heidelberg (2007)

[1] Available at `http://i12www.ira.uka.de/~bweiss/keyheap/`

3. Cheon, Y., Leavens, G.T., Sitaraman, M., Edwards, S.H.: Model variables: cleanly supporting abstraction in design by contract. Software—Practice and Experience 35(6), 583–599 (2005)
4. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Communications of the ACM 18(8), 453–457 (1975)
5. Harel, D., Kozen, D., Tiuryn, J.: Dynamic Logic. MIT Press, Cambridge (2000)
6. Hoare, C.A.R.: Proof of correctness of data representations. Acta Informatica 1, 271–281 (1972)
7. Kassios, I.T.: Dynamic frames: Support for framing, dependencies and sharing without restrictions. In: Misra, J., Nipkow, T., Karakostas, G. (eds.) FM 2006. LNCS, vol. 4085, pp. 268–283. Springer, Heidelberg (2006)
8. Leavens, G.T., Leino, K.R.M., Müller, P.: Specification and verification challenges for sequential object-oriented programs. Formal Aspects of Computing 19(2), 159–189 (2007)
9. Leino, K.R.M.: Toward Reliable Modular Programs. PhD thesis, California Institute of Technology (1995)
10. Leino, K.R.M.: Specification and verification of object-oriented software. Lecture Notes, Marktoberdorf International Summer School (2008)
11. Leino, K.R.M., Poetzsch-Heffter, A., Zhou, Y.: Using data groups to specify and check side effects. In: PLDI 2002, pp. 246–257. ACM Press, New York (2002)
12. Liskov, B., Wing, J.M.: A behavioral notion of subtyping. ACM Transactions on Programming Languages and Systems 16(6), 1811–1841 (1994)
13. McCarthy, J.: Towards a mathematical science of computation. In: Information Processing 1962, pp. 21–28 (1963)
14. Meyer, B.: Applying "design by contract". Computer 25(10), 40–51 (1992)
15. Müller, P., Poetzsch-Heffter, A., Leavens, G.T.: Modular invariants for layered object structures. Science of Computer Programming 62(3), 253–286 (2006)
16. Parkinson, M.J., Bierman, G.M.: Separation logic and abstraction. In: POPL 2005, pp. 247–258. ACM Press, New York (2005)
17. Poetzsch-Heffter, A.: Specification and verification of object-oriented programs. Habilitationsschrift, Technische Universität München (1997)
18. Rümmer, P.: Sequential, parallel, and quantified updates of first-order structures. In: Hermann, M., Voronkov, A. (eds.) LPAR 2006. LNCS (LNAI), vol. 4246, pp. 422–436. Springer, Heidelberg (2006)
19. Schmitt, P.H., Ulbrich, M., Weiß, B.: Dynamic frames in Java dynamic logic: Formalisation and proofs. Technical Report 2010-11, Department of Computer Science, Karlsruhe Institute of Technology (2010)
20. Smans, J., Jacobs, B., Piessens, F.: Implicit dynamic frames: Combining dynamic frames and separation logic. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 148–172. Springer, Heidelberg (2009)
21. Smans, J., Jacobs, B., Piessens, F., Schulte, W.: An automatic verifier for java-like programs based on dynamic frames. In: Fiadeiro, J.L., Inverardi, P. (eds.) FASE 2008. LNCS, vol. 4961, pp. 261–275. Springer, Heidelberg (2008)
22. Stenzel, K.: A formally verified calculus for full java card. In: Rattray, C., Maharaj, S., Shankland, C. (eds.) AMAST 2004. LNCS, vol. 3116, pp. 491–505. Springer, Heidelberg (2004)

A Refinement Methodology for Object-Oriented Programs*

Asma Tafat[1], Sylvain Boulmé[2], and Claude Marché[1,3]

[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 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 code 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.* In particular, there are many efforts to design a notion of *behavioral subtyping* constraining the subtyping relation of the programming language such that *Liskov substitution principle* is satisfied [23]: *Let* $\phi(x)$ *be a property provable about objects* x *of type* T*. Then* $\phi(y)$

* This work is partly supported by INRIA Collaborative Research Action (ARC) "CeProMi", `http://www.lri.fr/cepromi/`

should be true for objects y of type S where S is a subtype of T. Of course, such a notion of behavioral subtyping must also be expressive enough to allow the usual patterns of OO programming. This paper proposes a notion of behavioral subtyping, allowing sharing of objects, like in the observer pattern.

Actually, Liskov's substitution principle is very similar to the substitutability property of refinement approaches. Hence 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. The substitutability property ensures 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 the 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 [24], 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]. 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 [27] for a full one)

```
type bag<X>;
constant emptybag: bag<X>;
function singleton: X -> bag<X>;
function union: bag<X>, bag<X> -> bag<X>;
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);
function card: bag<X> -> integer;
function sumbag: bag<real> -> real;
...
```

2.2 Refinement

Refinement calculus [24,2] 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 1. Morgan's calculator [24] 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 $values : \mathtt{bag}(\mathbb{R})$
init $values \leftarrow \emptyset$;
op add($x : \mathbb{R}$):void =
  $values \leftarrow values \cup \{x\}$;
op mean():$\mathbb{R}$ =
  pre $values \neq \emptyset$;
  result $\leftarrow \dfrac{\mathtt{sumbag}(values)}{\mathtt{card}(values)}$;
```

```
var $count : \mathbb{N}$
var $sum : \mathbb{R}$
invariant $sum = \mathtt{sumbag}(values) \wedge$
          $count = \mathtt{card}(values)$;
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 a clients point of view. For any reachable state c, a satisfying I in the execution of a given client code, there exist abstract values a' such that I is still satisfied. For instance, in 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 2 (Calculator continued). The VC for the add operation is

$$\begin{array}{l}\forall count, sum, values, x;\ (sum = \mathtt{sumbag}(values) \wedge count = \mathtt{card}(values)) \Rightarrow \\ \quad \exists values'; values' = values \cup \{x\} \wedge \\ \qquad (sum + x = \mathtt{sumbag}(values') \wedge count + 1 = \mathtt{card}(values'))\end{array}$$

which is a logical consequence of our axiomatization of bags.

2.3 Model Fields

Model fields have been introduced by Leino [19] as abstract representations of object states. Syntactically, a *model field* [13] 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, 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 [13] : the latter can be directly assigned in implementations.

Example 3. 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;

  /*@ requires a.value >= 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 is the coupling invariant, relating the concrete fields (like `euros` and `cents` above) to the model field (`value` above), 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 $o.f_1 \ldots f_n.g$ only if $f_1, \ldots, f_n$ 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 the `inv` field into a class name: "$o.inv = C$" means that object o satisfies invariant of all superclasses of C (C included). Packing and unpacking 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 the next section (see also [27]), 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.$f_1 \ldots f_n.g$" 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 "$\tau\ m(x_1 : \tau_1, \ldots, x_n : \tau_n)$", a body, and a *contract* defined as:
  - a pre-condition $Pre_m(this, x_1, \ldots, x_n)$
  - a post-condition, $Post_m(this, x_1, \ldots, x_n, result)$ 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 belong to $locs$, is unchanged in the post-state.
- a set of constructors with a signature $C(x_1 : \tau_1, \ldots, x_n : \tau_n)$, 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 an 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 the 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: $o.inv = T \wedge Inv_T[this \mapsto o] \wedge (\textbf{old}(P))[v_i \mapsto o.M_i] \wedge$
    $\bigwedge_{f \in Comp_T} o.f \neq null \Rightarrow o.f.committed$

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 4. 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:

$$\begin{array}{l} this \neq null \wedge this.inv = Object \wedge \\ \exists v,\ this.sum = sumbag(v) \wedge this.count = card(v) \wedge \\ \quad v = union(this.values, singleton(x)) \end{array}$$

Hence, notice that the weakest precondition of `add` is thus very similar formula to the VC of the refinement given in Example 2.

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.

```
class SimpleCalc {
  //@ model bag<real> values;
  private int count;
  private double sum;
  //@ invariant sum == sumbag(values) && count == card(values);

  /*@ assigns \nothing;
    @ ensures inv == v \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

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 [27] 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

```
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

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 1. We would like to mimic this example in Java by splitting class `SimpleCalc` of Fig. 1 into two classes: first, 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. 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 datagroup 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.

```
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

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 [25]), 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 with the *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 update 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 [27]). 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 ;
  }
}
```

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 for 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 states of observers in one datagroup of `Subject`. However, the datagroups discipline (with the use of *pivot fields* [22,27]) would then prevent access to observers from outside of `Subject`, which is 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.[1] 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 Work and Perspectives

In 2003, Cheon et al. [13] propose foundations for the model fields in JML, 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,

[1] 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 packed in a dummy state before the call to `register`.

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 the context of deductive verification instead, JML also provides non-deterministic coupling relations via `\such_that` clauses. In 2003, Breunesse and Poll [10] explore four different interpretations of these clauses. 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 replacing 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

$$this.values = union(old(this.values), singleton(x))$$

because the latter is not the only bag b which has the given sum and cardinal. In other words, the Leino-Müller approach [20] can only deal with deterministic coupling invariants, which impose only one possible value for model fields from the values of the concrete fields.

Our methodology for refinement has several original aspects: 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 [30] 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.

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 (more precisely, ownership constructs are parsed and typechecked by ESC/Java2, but are not taken into account in the VC generation).

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* [26], *dynamic frames*, or region-based access control [28,29,3].

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 Urribarrì, Christine Paulin and others CeProMi members for their fruitful discussions on this work.

References

1. Abrial, J.-R.: The B-Book, assigning programs to meaning. Cambridge University Press, Cambridge (1996)
2. Back, R.-J., von Wright, J.: Refinement Calculus: A Systematic Introduction. Springer, Heidelberg (1998)
3. Banerjee, A., Naumann, D.A., Rosenberg, S.: Regional logic for local reasoning about global invariants. In: Ryan, M. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 387–411. Springer, Heidelberg (2008)
4. Barnett, M., DeLine, R., Fähndrich, M., Leino, K.R.M., Schulte, W.: Verification of object-oriented programs with invariants. Journal of Object Technology 3(6), 27–56 (2004)
5. Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006)
6. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# Programming System: An Overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 49–69. Springer, Heidelberg (2005)
7. Baudin, P., Filliâtre, J.-C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language (2008), `http://frama-c.cea.fr/acsl.html`
8. Behm, P., Benoit, P., Faivre, A., Meynadier, J.-M.: Météor: A successful application of B in a large project. In: Woodcock, J.C.P., Davies, J. (eds.) FM 1999. LNCS, vol. 1708, pp. 369–387. Springer, Heidelberg (1999)
9. Boulmé, S., Potet, M.-L.: Interpreting invariant composition in the B method using the spec# ownership relation: A way to explain and relax B restrictions. In: Julliand, J., Kouchnarenko, O. (eds.) B 2007. LNCS, vol. 4355, pp. 4–18. Springer, Heidelberg (2006)

10. Breunesse, C.-B., Poll, E.: Verifying JML specifications with model fields. In: FTfJP 2003 (2003)
11. Burdy, L., Cheon, Y., Cok, D., Ernst, M., Kiniry, J., Leavens, G.T., Leino, K.R.M., Poll, E.: An overview of JML tools and applications. International Journal on Software Tools for Technology Transfer (2004)
12. Charles, J.: Adding native specifications to JML. In: FTfJP 2006 (2006)
13. Cheon, Y., Leavens, G., Sitaraman, M., Edwards, S.: Model variables: cleanly supporting abstraction in design by contract. Softw. Pract. Exper. 35(6), 583–599 (2005)
14. Cok, D.R., Kiniry, J.R.: ESC/Java2: Uniting eSC/Java and JML. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 108–128. Springer, Heidelberg (2005)
15. Darvas, A.P.: Reasoning About Data Abstraction in Contract Languages. PhD thesis, ETH Zurich (2009)
16. Filliâtre, J.-C., Marché, C.: Multi-prover verification of C programs. In: Davies, J., Schulte, W., Barnett, M. (eds.) ICFEM 2004. LNCS, vol. 3308, pp. 15–29. Springer, Heidelberg (2004)
17. Filliâtre, J.-C., Marché, C.: The why/Krakatoa/Caduceus platform for deductive program verification. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 173–177. Springer, Heidelberg (2007)
18. Leavens, G.T., Leino, K.R.M., Müller, P.: Specification and verification challenges for sequential object-oriented programs. In: Formal Aspects of Computing (2007)
19. Leino, K.R.M.: Data groups: Specifying the modification of extended state. In: OOPSLA 1998, pp. 144–153 (1998)
20. Leino, K.R.M., Müller, P.: A verification methodology for model fields. In: Sestoft, P. (ed.) ESOP 2006. LNCS, vol. 3924, pp. 115–130. Springer, Heidelberg (2006)
21. Leino, K.R.M., Nelson, G.: Data abstraction and information hiding. ACM Trans. Prog. Lang. Syst. 24(5), 491–553 (2002)
22. Leino, K.R.M., Poetzsch-Heffter, A., Zhou, Y.: Using data groups to specify and check side effects. In: PLDI 2002. ACM, New York (2002)
23. Liskov, B., Wing, J.M.: A behavioral notion of subtyping. ACM Trans. Program. Lang. Syst. 16(6), 1811–1841 (1994)
24. Morgan, C.: Programming from specifications, 2nd edn. Prentice Hall International (UK) Ltd., Englewood Cliffs (1994)
25. Parkinson, M.: Class invariants: The end of the road. In: IWACO 2007 (2007), http://www.cs.purdue.edu/homes/wrigstad/iwaco/
26. Reynolds, J.C.: Separation logic: a logic for shared mutable data structures. In: 17h Annual IEEE Symposium on Logic in Computer Science. IEEE Comp. Soc. Press, Los Alamitos (2002)
27. Tafat, A., Boulmé, S., Marché, C.: A refinement approach for correct-by-construction object-oriented programs. Technical Report RR-7310, INRIA (2010)
28. Talpin, J.-P., Jouvelot, P.: Polymorphic type, region and effect inference. Journal of Functional Programming 2(3), 245–271 (1992)
29. Tofte, M., Talpin, J.-P.: Region-based memory management. Information and Computation 132(2), 109–176 (1997)
30. Zee, K., Kuncak, V., Rinard, M.: Full functional verification of linked data structures. In: PLDI 2008, pp. 349–361. ACM Press, New York (2008)

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 using invariants which can be used to deal with loops during the verification.

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 intermediate verification language that covers the essential features of established intermediate languages and is close to Boogie [10]. We present a program logic in the style of first-order dynamic logic (DL).

DL is a program logic which embeds pieces of code within formulas. In its original presentation [9] by Harel et al., a code fragment π in a structural language gives rise to a modality $[\pi]$ which can be used as a 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 $\psi \to [\pi]\phi$ 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 the 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 for finding mismatches between specification and implementation, but also particularily valuable when experimenting with new modelling or translation techniques. Other approaches use weakest precondition (wp) calculi to automatically generate first order verification conditions. 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.

An intermediate verification language provides – like the intermediate representation for compilers – a common verification platform for different programming languages and specification techniques. If it is general enough, many programming languages can be translated to it, thereby decoupling the semantics of the source language from the actual verification process.

The design of the present 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.

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. An overview of related work in Sect. 5 and conclusions in Sect. 6 wrap up the paper.

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 [11] 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 instance, if a program contains a division expression $1/x$, we need to verify that x is different from 0 to ensure its correctness. This check cannot be postponed to the after state of the entire code, but needs to performed in the state in which the expression is evaluated. This can be addressed by inserting an assertion at the appropriate place into the intermediate code.

2.1 Syntax

USDL is an extension of first order logic with two additional modal 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* $\Sigma = (\mathrm{Var}, \mathrm{Fct}, \mathrm{PVar}, \mathrm{Prd}, \alpha)$ *is a 5-tuple with*

$$
\begin{array}{lll}
\text{Formula} & ::= & \text{Formula} \;(\, \wedge \mid \vee \mid \rightarrow \,)\; \text{Formula} \\
 & \mid & \neg\ \text{Formula} \mid \textbf{true} \mid \textbf{false} \\
 & \mid & (\forall \mid \exists)\, \mathit{Var}\ \textbf{.}\ \text{Formula} \\
 & \mid & \mathit{Prd} \quad\mid\quad \mathit{Prd}\ \textbf{(}\ \text{TermList}\ \textbf{)} \qquad (*) \\
 & \mid & \textbf{\{}\ \text{Update}\ \textbf{\}}\ \text{Formula} \\
 & \mid & [\ \mathit{NaturalNumber}\ \textbf{;}\ \text{Program}\ ] \\
 & \mid & [[\ \mathit{NaturalNumber}\ \textbf{;}\ \text{Program}\ ]] \\
\text{Term} & ::= & \mathit{Var} \quad\mid\quad \mathit{Fct} \quad\mid\quad \mathit{Fct}\ \textbf{(}\ \text{TermList}\ \textbf{)} \qquad (*) \\
 & \mid & \textbf{\{}\ \text{Update}\ \textbf{\}}\ \text{Term} \\
\text{TermList} & ::= & \text{Term} \quad\mid\quad \text{TermList}\ \textbf{,}\ \text{TermList} \\
\text{Update} & ::= & \mathit{PVar}\ \textbf{:=}\ \text{Term} \quad\mid\quad \text{Update} \parallel \text{Update} \\
\text{Program} & ::= & \text{Statement} \quad\mid\quad \text{Program}\ \textbf{,}\ \text{Program} \\
\text{Statement} & ::= & \mathit{PVar}\ \textbf{:=}\ \text{Term} \\
 & \mid & \textbf{assert}\ \text{Formula} \qquad\qquad \mid \textbf{assume}\ \text{Formula} \\
 & \mid & \textbf{havoc}\ \mathit{PVar} \qquad\qquad\quad\ \mid \textbf{goto}\ \mathit{NaturalNumber} \\
 & \mid & \textbf{goto}\ \mathit{NaturalNumber}\ \mathit{NaturalNumber}
\end{array}
$$

(*) if the length of the term list coincides with the arity of the symbol

Fig. 1. Formulas, Terms and Programs

- Var*: the set of logical variable symbols*
- Fct*: the non-empty set of function symbols*
- PVar $\subseteq$ 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 $pos \in \mathrm{Prd}$ and a unary function symbol $suc \in \mathrm{Fct}$. The expression

$$[0; \mathsf{goto}\ 1\ 4, \mathsf{assume}\ \neg pos(x), x := suc(x), \mathsf{goto}\ 0, \\ \mathsf{assume}\ pos(x), \mathsf{assert}\ pos(x)] \quad (1)$$

is then a valid atomic program formula in Form_Σ. In Ex. 1 we show how this formula can be used for the symbolic execution of the contained program.

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 include statements located before the active statement in the modalities. This is because goto statements may refer to any position in the program, before or after the current one. We employ an explicit program counter indicating 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 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 syntactically describes a change of the evaluation state. Applying an update to an evaluation context overrides the interpretation function. Therefore, in the upcoming definition and for an update $c_1 := t_1 \| \dots \| c_n := t_n$ and an interpretation I we use the notation

$$I^{c_1:=t_1\|\dots\|c_n:=t_n} := ((I_{c_1}^{val_{I,\beta}(t_1)})\dots)_{c_n}^{val_{I,\beta}(t_n)}$$

to denote the interpretation in which the symbols $c_1, \dots, c_n$ 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* $\mathrm{val}_{I,\beta} : \mathit{Term}_\Sigma \to \mathcal{D}$ *is defined by:*

- $\mathrm{val}_{I,\beta}(x) = \beta(x)$ *if* $x \in \mathrm{Var}$,
- $\mathrm{val}_{I,\beta}(f(t_1, \dots, t_k)) = I(f)(\mathrm{val}_{I,\beta}(t_1), \dots, \mathrm{val}_{I,\beta}(t_k))$ *if* $f \in \mathrm{Fct}$ *with* $\alpha(f) = k$ *and* $t_1, \dots, t_k \in \mathit{Term}_\Sigma$,
- $\mathrm{val}_{I,\beta}(\{\mathcal{U}\}t) = val_{I^\mathcal{U},\beta}(t)$

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. It may appear counter-intuitive that in Def. 8, the semantics of assert and assume statements seem identical. The difference is that a trace is considered *successful* if it fails at an assumption but *unsuccessful* for a failed assertion.

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*** $R_\pi(s) =$
$c := t$		$\{(I_c^{\mathrm{val}_{I,\beta}(t)}, n+1)\}$
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 $(s_0, s_1, \dots, s_r)$ (or $(s_0, s_1, \dots)$ 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 $(s_0, s_1, \dots, s_r)$ 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).

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:*

- $I, \beta \models \text{true}$ *and* $I, \beta \not\models \text{false}$
- $I, \beta \models \phi\,(\wedge|\vee|\rightarrow)\,\psi$ *iff* $I, \beta \models \phi$ *and/or/implies* $I, \beta \models \psi$.
- $I, \beta \models (\forall|\exists)x.\phi$ *iff* $I, \beta^d_x \models \phi$ *for every/some* $d \in \mathcal{D}$.
- $I, \beta \models p(t_1, \dots, t_k)$ *iff* $(\text{val}_{I,\beta}(t_1), \dots, \text{val}_{I,\beta}(t_k)) \in I(p)$ *for a predicate symbol* $p \in \text{Prd}$ *with* $\alpha(p) = k$ *and* $t_1, \dots, t_k \in Term_\Sigma$.
- $I, \beta \models \{\mathcal{U}\}\phi$ *iff* $I^{\mathcal{U}}, \beta \models \phi$
- $I, \beta \models [n; \pi]$ *iff every maximal finite trace* $(I, n), \dots, (I_k, n_k)$ *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 $I(pos) = \mathbb{N}$. 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 [9]. 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.

Hence, we can 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

$$[0; \mathsf{while}\ \neg pos(x)\ \mathsf{do}\ x := suc(x)\ \mathsf{end}; \mathsf{assert}\ pos(x)] \tag{2}$$

using such a macro for the while-do-end loop. This formula embeds in a formula the meaning of the Hoare triple $\{\text{true}\}\mathsf{while}\ \neg pos(x)\ \mathsf{do}\ x := suc(x)\ \mathsf{end}\{pos(x)\}$.

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 1 (Symbolic execution). *The following rules are sound rewrite rules for the symbolic execution of unstructured programs.*

$$\pi[n] = c := v \implies [n;\pi] \rightsquigarrow \{c := v\}[n+1;\pi] \tag{3}$$
$$\pi[n] = \mathsf{havoc}\ c \implies [n;\pi] \rightsquigarrow \forall x.\{c := x\}[n+1;\pi] \tag{4}$$
$$\pi[n] = \mathsf{goto}\ m \implies [n;\pi] \rightsquigarrow [m;\pi] \tag{5}$$
$$\pi[n] = \mathsf{goto}\ m\ k \implies [n;\pi] \rightsquigarrow [m,\pi] \wedge [k;\pi] \tag{6}$$
$$\pi[n] = \mathsf{assume}\ \phi \implies [n;\pi] \rightsquigarrow \phi \rightarrow [n+1;\pi] \tag{7}$$
$$\pi[n] = \mathsf{assert}\ \phi \implies [n;\pi] \rightsquigarrow \phi \wedge [n+1;\pi] \tag{8}$$

Proof. The soundness proofs for these rules are straightforward. We exemplarily provide them for (7) and (8). 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: If $I, \beta \not\models \phi$, then $R_\pi(I, n) = \emptyset$ and the only trace beginning in (I, n) ends in an assume statement and, hence, *is* successful. If $I, \beta \models \phi$, 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 in an assert statement and, hence, *is not* successful. The other case depends again on the traces 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 one 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 1. *We obtain sound rewrite rules if we replace every occurrence of a modality* $[n;\pi]$ *by the corresponding terminating counterpart* $[[n;\pi]]$ *in* (3)–(8)*.*

Example 1. Let us now reconsider formula (1) which is $[0,\pi]$ for the program

$$\pi = \big(\mathsf{goto}\ 1\ 4; \mathsf{assume}\ \neg pos(x); x := suc(x); \mathsf{goto}\ 0; \mathsf{assume}\ pos(x); \mathsf{assert}\ pos(x)\big)\ .$$

By repeatedly applying the calculus rules (3)–(8), we can execute the program, statement by statement resulting in the following chain of equivalent formulas.

$$\begin{aligned} [0;\pi] \quad & \rightsquigarrow \quad [1;\pi] \wedge [4;\pi] \\ \overset{2\times}{\rightsquigarrow} \ & (\neg pos(x) \to [2;\pi]) \wedge (pos(x) \to [5;\pi]) \\ \overset{2\times}{\rightsquigarrow} \ & (\neg pos(x) \to \{x := suc(x)\}[3;\pi]) \wedge (pos(x) \to (pos(x) \wedge [6;\pi])) \\ \overset{2\times}{\rightsquigarrow} \ & (\neg pos(x) \to \{x := suc(x)\}[0;\pi]) \wedge (pos(x) \to (pos(x) \wedge \mathsf{false} \to [7;\pi])) \end{aligned}$$

Since in the last step, the index 6 is outside the index range of π, $[6,\pi]$ is equivalent to false $\to [7,\pi]$ which is obviously true. $[3,\pi]$ is the same as $[0,\pi]$ and a loop is entered. The next section covers how we deal with such situations. This is a very simple example. In larger, more complex programs, one can learn more about the verification condition if one can interact during its generation.

4 Invariant Rules

The rewrite rules in Thm. 1 and Cor. 1 allow the symbolic execution of 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 can canonically be combined to a rule with termination and context preservation.

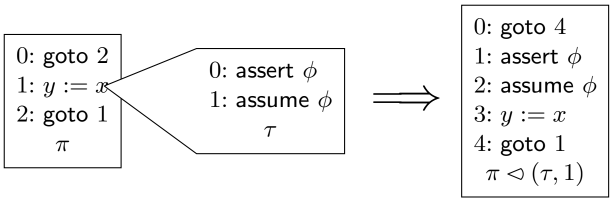


Fig. 2. Example of a program insertion

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 traces 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.

$$\mathit{off}_m^k(a) = \begin{cases} a & \text{if } a \le m \\ a + k & \text{otherwise} \end{cases}$$

We also apply off_m^k to statements. Here, it operates only on the target indices of goto statements and behaves like the identity function on all other statements.

Definition 10 (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*

$$\big(\pi \lhd (\tau, m)\big)[i] = \begin{cases} \mathit{off}_m^{len(\tau)}(\pi[i]) & \text{for } i < m \\ \tau[i - m] & \text{for } m \le i < m + len(\tau) \\ \mathit{off}_m^{len(\tau)}(\pi[i - len(\tau)]) & \text{for } m + len(\tau) \le i \end{cases} .$$

τ is not subject to an offset correction since the programs we use for insertion here will not contain goto statements.

Fig. 2 shows a sample program insertion. The program $\tau = (\text{assert } \phi; \text{assume } \phi)$ is inserted into the program $\pi = (\text{goto } 2; y := x; \text{goto } 1)$ at position 1. Please note that in statement 4 : 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_m^k, a trace for π which does not pass through the insertion point m induces a trace for the program after insertion (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 1. For any trace $(I_0, k_0), \ldots, (I_r, k_r)$ with $k_i \neq n$ for $0 < i \leq r$, the sequence $(I_0, k'_0), \ldots, (I_r, k'_r)$ with $k'_i = \mathit{off}_n^{len(\tau)}$ is a trace for $\pi \lhd (\tau, n)$.

The rules we develop in this section are inference rules for a *sequent calculus.* A sequent is of the form $\Gamma \vdash \Delta$ with *antecedent* Γ and the *succedent* Δ finite sets of formulas. The sequent 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 keyword "while". 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 points at which an invariant rule should be applied.

4.2 Simple Invariant Rule

The general idea in the upcoming invariant rules is to change a program in such a way that a loop becomes dissected. At the beginning of the loop, an invariant is assumed which has to be asserted whenever the initial statement is reached again during symbolic execution. For that purpose we insert the statements (assert ϕ; assume false) at the current position.

Theorem 2. *The rule*

$$\frac{\Gamma \vdash \{\mathcal{U}\}\psi, \Delta \qquad \psi \vdash [n+2; \rho_1]}{\Gamma \vdash \{\mathcal{U}\}[n; \pi], \Delta}$$

with $\rho_1 = \pi \lhd ((\text{assert } \psi; \text{assume false}), n)$ *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. 4.

This rule is similar to the invariant rule for a dynamic logic for a simple 'while'-language. One difference is that, here, we have *two* rather than *three* 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: It proves the absence of assertion violations after the loop (the 'use case' of ψ), and it ensures that the loop body preserves ψ establishing it as an invariant.

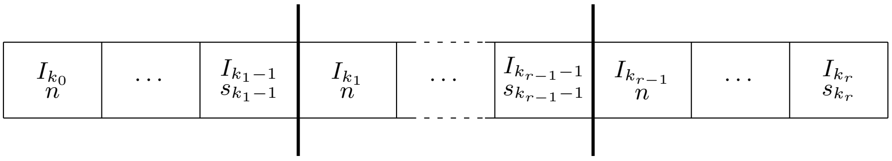


Fig. 3. Chopping a trace into subtraces

Proof. We can without loss of generality[1] assume that $\varDelta = \emptyset$. Moreover, we may assume that (A) $\bigwedge\varGamma \to \{\mathcal{U}\}\psi$ and (B) $\psi \to [n+2;\rho_1]$ are valid formulas. For an arbitrary interpretation[2] I, we need to show that $I \models \bigwedge\varGamma \to \{\mathcal{U}\}[n;\pi]$. If $I \not\models \bigwedge\varGamma$, the proof is completed. Thus, let $I \models \bigwedge\varGamma$. 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 $0 \le i < r$, 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 $0 \le i < r$. For $I_{k_0} (= I^{\mathcal{U}})$, this is a simple consequence of the validity of (A). Now, we assume that $I_{k_i} \models \psi$ for some $0 \le i < r-1$.

For the trace $(I_{k_i}, n), \dots, (I_{k_{i+1}-1}, s_{k_{i+1}-1})$, apart from the first state, no state is in statement n: it matches the requirements of Prop. 1, and, thus, we know that $(I_0, n+2), \dots, (I_{k_{i+1}-1}, \mathit{off}^2_n(s_{k_{i+1}-1}))$ 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 is 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 $(I_{k_{r-1}}, n), \dots, (I_{k_r}, s_{k_r})$ 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. 1 again and obtain a trace $(I_{k_{r-1}}, n+2), \dots, (I_{k_r}, \mathit{off}^2_n(s_{k_r}))$ 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. 2 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

[1] There are first order inference rules that allow us to move the negation of all formulas in $\varDelta$ to the antecedent $\varGamma$.

[2] For the sake of readability, we leave variable assignments aside in this section.

relation. With the aid of this predicate symbol, we can formulate an invariant rule which includes termination.

Theorem 3. *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 $\rho_2 = \pi \lhd ((\mathsf{assert}\ \psi \wedge var \prec nc; \mathsf{assume}\ \text{false}), n)$ *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. 2 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. 2 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 $\big((I_{k_{r-1}}, n), \ldots\big)$ 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 $(v_1, v_2, \ldots)$ 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. 2 and 3 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 (4).

The rule follows the ideas of [3] where 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 11 (loop-reachable). *A statement m is called* loop-reachable *from n within a program π if there is a trace $(I_o, k_0), (I_1, k_1), \ldots$ 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

$$mod(n, \pi) := \left\{ c \;\middle|\; \begin{array}{l} \text{there are } m, c \text{ and } t \text{ s.t. } reach(n, m, \pi) \text{ and} \\ (\pi[m] = \mathsf{havoc}\ c \text{ or } \pi[m] = c := t) \end{array} \right\} \subseteq \text{PVar} \quad .$$

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. 2 and the concluding assumption corresponds to the formula ψ in the antecedent of the second premiss in Thm. 2. 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 4. *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 $mod(n; \pi) \subseteq \{r_1, \ldots, r_b\} \subseteq$ 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

$$\Gamma \vdash \forall x_1 . \ldots \forall x_b . \{r_1 := x_1 \| \ldots \| r_b := x_b\}(\psi \to [n + 2; \rho_1]) \; . \qquad (9)$$

For an interpretation I with $I \models \bigwedge \Gamma$, we know, because of the validity of the premiss, that I makes the formula in (9) true. 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. 2.

In particular we have $I_{k_{r-1}} \models [n+2; \rho_1]$ for which we saw in the proof of Thm. 2 that it implies that the entire trace is successful. $\square$

5 Related Work

While some verification tools (e.g., [2], [14]) take advantage of the greater transparency of source code verification, most employ a special-purpose intermediate language. The *Why* language [8] and the Forge Intermediate Representation (FIR) [7], for instance, are used as the target languages by various tools. Also, verification using the low level virtual machine (LLVM) format is a topic of ongoing research [13]. Boogie [6,10] is the most popular intermediate language and is used as target language for various object-oriented and imperative source code languages (incl. C$^\#$, Java, Dafny, Eiffel, ...). Barnett and Leino [1] describe how the Boogie verification condition generator breaks up loops using invariants in a fashion similar to this work. In [12], Quigley defines a Hoare-style calculus for Java bytecode. 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. Burdy and Pavlova describe a wp-calculus for Java bytecode in [5]. Therein, loops are resolved by a code modification rendering the control flow acyclic prior to the wp-calculation. HOL/Boogie [4], 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.

6 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 [9] 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.

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. Barnett, M., Leino, K.R.M.: Weakest-precondition of unstructured programs. In: Ernst, M.D., Jensen, T.P. (eds.) PASTE 2005, pp. 82–87. ACM Press, New York (2005)
2. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software. LNCS (LNAI), vol. 4334. Springer, Heidelberg (2007)
3. Beckert, B., Schlager, S., Schmitt, P.H.: An improved rule for while loops in deductive program verification. In: Lau, K.-K., Banach, R. (eds.) ICFEM 2005. LNCS, vol. 3785, pp. 315–329. Springer, Heidelberg (2005)
4. Böhme, S., Leino, K.R.M., Wolff, B.: HOL-boogie — an interactive prover for the boogie program-verifier. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 150–166. Springer, Heidelberg (2008)
5. Burdy, L., Pavlova, M.: Java bytecode specification and verification. In: Liebrock, L.M. (ed.) SAC 2006. ACM, New York (2006)
6. DeLine, R., Leino, K.R.M.: BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, Redmond (2005)
7. Dennis, G., Chang, F.S.-H., Jackson, D.: Modular verification of code with SAT. In: Pollock, L.L., Pezzè, M. (eds.) ISSTA 2006, pp. 109–120. ACM Press, New York (2006)
8. Filliâtre, J.-C., Marché, C.: The why/Krakatoa/Caduceus platform for deductive program verification. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 173–177. Springer, Heidelberg (2007)
9. Harel, D., Kozen, D., Tiuryn, J.: Dynamic Logic. MIT Press, Cambridge (2000)
10. Leino, K.R.M.: This is Boogie 2 (2008), Manuscript KRML 178, `http://research.microsoft.com/~leino/papers.html`
11. Leino, K.R.M., Rümmer, P.: A polymorphic intermediate verification language: Design and logical encoding. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 312–327. Springer, Heidelberg (2010)
12. Quigley, C.L.: A programming logic for java bytecode programs. In: Basin, D., Wolff, B. (eds.) TPHOLs 2003. LNCS, vol. 2758, pp. 41–54. Springer, Heidelberg (2003)
13. Sinz, C., Falke, S., Merz, F.: A precise memory model for low-level bounded model checking. In: Huuck, R., Klein, G., Schlich, B. (eds.) SSV 2010 (2010)
14. Stenzel, K.: Verification of Java Card Programs. PhD thesis, University of Augsburg (2005)

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 and the JUnit testing framework on which it is based. 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 JMLUnitNG, a TestNG-based successor to JMLUnit that can automatically generate and execute millions of tests, using supplied test data of only primitive types, without consuming excessive amounts of memory. We also present a comparison of test coverage between JMLUnitNG and 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 (SUT). 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 and the JUnit testing framework on which it is based. After providing some background information about unit testing, JML, and JMLUnit, we describe the limitations of JMLUnit for testing complex systems. We then address these limitations with JMLUnitNG, a successor to JMLUnit based on the TestNG testing framework. Finally, we demonstrate our improvements using coverage results from tests generated by both JMLUnit and JMLUnitNG. The goals of this work are to make automated unit test generation for JML-annotated Java programs more effective and easier for developers 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 given 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.

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

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.

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; 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

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

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.

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` parameters `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 using reflection 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; this makes it impossible to save memory by parameterizing the tests.

In order to enable lazy parameter list generation, we replace the underlying JUnit engine used by JMLUnit 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

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

detail as we choose. As distributed, TestNG does record every test execution in 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 `SkipExceptions`. 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 Comparison of JMLUnit and JMLUnitNG

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;

[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.

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

		Covered Blocks		**% Covered**	
Class	**Total Blocks**	**Orig**	**New**	**Orig**	**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 2. Results for Chess classes with JMLUnit (Orig) and JMLUnitNG (New)

		Covered Blocks		**% Covered**	
Class	**Total Blocks**	**Orig**	**New**	**Orig**	**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

`Candidate` stores information about a single candidate for office; `KiesLijst` 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

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

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

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

the chess classes, while JMLUnit does the opposite—is due to the fact that the constructors for the chess classes have significantly less 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 2002, Brno, Czech Republic (August 2002)
2. Beckert, B., Hähnle, R., Schmitt, P.H.: Verification of Object-Oriented Software. The KeY Approach. LNCS (LNAI), vol. 4334. Springer, Heidelberg (2007)
3. Bouquet, F., Dadeau, F., Legeard, B., Utting, M.: JML-testing-tools: A symbolic animator for JML specifications using CLP. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 551–556. Springer, Heidelberg (2005)
4. Bourdonov, I.B., Kossatchev, A.S., Kuliamin, V.V., Petrenko, A.K.: UniTesK test suite architecture. In: Eriksson, L.-H., Lindsay, P.A. (eds.) FME 2002. LNCS, vol. 2391, p. 77. Springer, Heidelberg (2002)
5. Brucker, A.D., Wolff, B.: Interactive testing with HOL-testGen. In: Grieskamp, W., Weise, C. (eds.) FATES 2005. LNCS, vol. 3997, pp. 87–102. Springer, Heidelberg (2006)
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 (February 2005)
7. Cheon, Y., Leavens, G.T.: A simple and practical approach to unit testing: The JML and jUnit way. In: Deng, T. (ed.) ECOOP 2002. LNCS, vol. 2374, pp. 231–255. Springer, Heidelberg (2002)
8. Deng, X., Robby, H.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), Windsor, UK, pp. 3–12 (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), Switzerland (February 2007)
11. King, J.C.: Symbolic execution and program testing. Communications of the ACM 19(7), 385–394 (1976)
12. Kiniry, J.R., Morkan, A.E., Cochran, D., Fairmichael, F., Chalin, P., Oostdijk, M., Hubbers, E.: The KOA remote voting system: A summary of work to date. In: Montanari, U., Sannella, D., Bruni, R. (eds.) TGC 2006. LNCS, vol. 4661, pp. 244–262. Springer, Heidelberg (2007)
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: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2002. LNCS, vol. 2852, pp. 262–284. Springer, Heidelberg (2003)
14. Meyer, B.: Object-Oriented Software Construction, 2nd edn. Prentice-Hall, Englewood Cliffs (1988)
15. Peters, D.K., Parnas, D.L.: Using test oracles generated from program documentation. IEEE Transactions on Software Engineering 24(3), 161–173 (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 (1997)

Author Index